Competitive Buffer Management for Packets with Latency Constraints

Alex Davydow[a], Pavel Chuprikov[b], Sergey I. Nikolenko[c,*], Kirill Kogan[d]

[a]*Harbour Space University, Carrer de Rosa Sensat, 9, 08005 Barcelona, Spain*
[b]*IMDEA Networks Institute, Av. Mar Mediterraneo, 22, 28918 Leganes, Madrid, Spain*
[c]*Harbour Space University, Carrer de Rosa Sensat, 9, 08005 Barcelona, Spain*
[d]*IMDEA Networks Institute, Av. Mar Mediterraneo, 22, 28918 Leganes, Madrid, Spain*

Abstract

Modern datacenters are increasingly required to deal with latency-sensitive applications. Incorporation of multiple traffic characteristics (e.g., packet values and required processing requirements) significantly increases the complexity of buffer management policies. In this context two major questions arise: how to represent the latency in desired objectives and how to provide guarantees for buffer management policies that would hold across a wide variety of traffic patterns. In this work, we consider a single queue buffering architecture, where every incoming packet is prepended with intrinsic value, required processing, and slack (offset from the arrival time during which this packet should be transmitted); the buffer size is implicitly bounded by slack values. Our goal is to maximize a total transmitted value (weighted throughput). In these settings, we study worst-case performance guarantees of the proposed online algorithms by means of competitive analysis whose effectiveness is compared versus an optimal clairvoyant offline algorithm. We show non-constant general lower bounds that hold for arbitrary slack values and for slacks that are *additively* separated from processing requirements; for the case of a *multiplicative* separation, we present a novel buffer management policy SPQ (stack with priority queue) and show that it is at most 3-competitive. Our theoretical results are supported by a comprehensive evaluation study on CAIDA traces.

Keywords: packets with deadlines, buffer management, competitive analysis

1. Introduction

Low latency is critical for interactive applications, and achieving desired latency is a challenging task. Modern applications are highly distributed; moreover, in order to complete a final result application-level operations consisting of many queries often should satisfy latency constraints [1, 2, 3].

Latency is certainly an important factor, but there are two primary ways to represent latency requirements for optimization problems that arise in the design of network elements. One approach is to include latency in the final objectives by actually optimizing for it; for example, a popular way to represent latency in the final objective is to minimize *average flow completion time* [4, 5]. This, however, requires prior knowledge about flow duration which may be too restrictive, and still leaves the problem of how to balance latency with other desired objectives such as throughput, fairness, and others.

Instead of optimizing latency explicitly, a different way to look at the problem is to satisfy latency constraints on the level of individual packets; this constraint-based approach allows to optimize additional objectives and does not require prior knowledge of a flow's duration. Originally, weighted throughput optimization for packets with associated intrinsic value and slack was introduced in [6] and called a *bounded-delay model*. In this model, the *slack* defines a latency constraint, i.e., the offset from the arrival time when a packet must be transmitted or is lost otherwise, and every successfully transmitted packet contributes its *intrinsic value* to the weighted throughput.

*Corresponding author

Email addresses: adavydow@gmail.com (Alex Davydow), pschuprikov@gmail.com (Pavel Chuprikov), snikolenko@gmail.com (Sergey I. Nikolenko), kirill.kogan@gmail.com (Kirill Kogan)

However, the model of [6] does not include another important characteristic: processing requirements per packet. Different packets at the same network element can require different processing; from simple forwarding on some of them to complex services such as deep packet inspection (DPI) or sophisticated virtual private network (VPN) services on the others; this characteristic has only recently become an object of study in admission control and scheduling decisions [7, 8]. However, none of these works apply latency constraints. Hence, the primary objective of this work is to incorporate all three characteristics: packet values, processing requirements, and latency constraints, studying the combined impact of the three characteristics on throughput optimization.

In real deployment scenarios, traffic patterns can vary greatly, and simple probabilistic assumptions on the properties of an incoming stream of packets may be violated in unpredictable ways. Moreover, in practice the choices of processing order and admission control logic are likely to be made at design time and often cannot be updated "on the fly" when conditions in the network environment change. Hence, in this work we mostly study buffer management policies from the point of view of their *worst-case* behaviour, aiming to provide a robust estimate on the settings that can handle all possible arrival patterns. In the field of online algorithms, and also specifically buffer management algorithms, worst-case guarantees are usually obtained via *competitive analysis* [9, 10]. Informally, *competitiveness* (also known as *competitive ratio*) is a number that shows how much worse an online algorithm may perform in the worst case (on the worst possible packet arrival sequence) compared to an offline clairvoyant optimal algorithm, i.e., an algorithm that knows the input sequence in advance, has unlimited computational resources, and hence can compute and apply the best possible decisions for every sequence of arriving packets. Thus, if, e.g., an algorithm is at most 2-competitive this means that it is at most twice worse than the optimal algorithm *on any arrival sequence*, a result that provides a uniform guarantee for all networking environments and without any assumptions regarding traffic patterns.

Formally, we consider a single-queue switch where a buffer is shared among all types of traffic, and arriving packets are characterized by three values: a packet $[\![w \mid v \mid s]\!]$ has required processing (work) w, value v that it contributes to the objective upon transmission, and slack s that shows how much time we have to process it (in s time slots after arrival, the packet will be automatically dropped and lost). We do not assume any specific traffic distribution but rather analyze our buffer management policies against adversarial traffic using competitive analysis. An online algorithm ALG is said to be *α-competitive* for some $\alpha \geq 1$ if for any arrival sequence σ the total value of packets transmitted by ALG is at least $1/\alpha$ times the total value transmitted in an optimal solution obtained by an offline clairvoyant algorithm, further denoted OPT. Note that a *lower bound* on the competitive ratio can be proven with a specific example while an *upper bound* represents a general statement that should hold over all possible inputs.

Our contributions in this work are as follows. We first introduce the formal setting (Section 3), consider several naturally arising algorithms for this setting based on priority queues, and show that in this setting with three different characteristics (values, processing requirements, and slacks) these priority queues have significantly different behaviours. Next, we establish *general* lower bounds on the competitiveness, i.e., bounds that hold for every deterministic online algorithm; in particular, not only there are no algorithms with constant competitiveness in this general model (Theorem 1) but also in case where there are no "real-time" packets, i.e., when every packet's allowed processing time (slack) exceeds its required processing time by an additive constant (Theorem 2); we also prove stronger results specifically for various modifications of priority queues (Theorems 3, 4, and 5). This negative part of our results may make constant competitiveness seem unattainable in this model. However, next we show that while an *additive* constraint on the slacks is not enough, a *multiplicative* one will do: we introduce a novel algorithm SPQ (stack with priority queue) and show that under the assumption that for every incoming packet $[\![w \mid v \mid s]\!]$ it holds that $s \geq cw$ for some constant c, SPQ has a constant upper bound on its competitive ratio; this is the main theoretical result of this paper (Theorem 6). We also show positive results for important special cases with bounded value (Theorem 9) and bounded work (Theorem 10). Finally, we provide a comprehensive evaluation study that compares various priority queues and the proposed SPQ algorithm on CAIDA traces [11] that confirms our theoretical findings and provides new insights.

The paper is organized as follows. In Section 2, we survey previous work in related buffer processing algorithms. Section 3 introduces a model that is used in this work. In Section 4, we propose several algorithms based on different processing orders, prove a general lower bound for all deterministic online algorithms, and show that all proposed algorithms have at least linear competitive ratio in the general case. Section 5 discusses a feasibility of a constant competitiveness and introduces a 3-competitive algorithm that is a main result of this paper, and Section 6 deals with two important special cases. In Section 7 we evaluate the proposed algorithms on CAIDA traces [11]. Section 8 concludes the paper. This work is a significantly extended version of the conference paper [12]; the most important additions to the version of [12] include new Theorems 2 and 4 a new Section 6, and a completely reworked evaluation study shown in Section 7.

2. Related Work

Our current work can be viewed as part of a larger research effort concentrated on studying competitive algorithms for management of various buffering architectures. Initiated in [13, 14], the competitive analysis for buffer management has received tremendous attention over the past decade. The bounded-delay model [6] has been shown to be effective for service scale-outs in [15]. Surveys by Goldwasser [16] and Nikolenko and Kogan [17] provide a comprehensive overview of this field.

Classical (decade-old) results in buffer management consider switches with multiple output queues and identical packets; in these settings, the LQD policy that pushes out a packet from the longest queue in case of congestion was proven to have constant competitive ratio for shared memory [18], while for separate queues constant competitiveness was achieved in [19].

The first considered characteristic was the *value*, i.e., how much the packet contributes to the objective function when it is transmitted. In this case, the PQ (priority queue) policy that processes most valuable packets first and pushes out least valuable packets is obviously optimal, so research concentrated on non-preemptive policies that cannot push out packets [20, 21] and preemptive policies with FIFO requirement, with a general lower bound of 1.419 for all deterministic online algorithms [22] and an upper bound of 2 for the greedy push-out algorithm [6]. The case of multiple separated queues with a uniform processing requirements (so the policy must select which queue to process a packet from) has been considered in [23], while the work [24] considers the case when each output queue processes a packet separately, and packets are assigned to a specific output port. Multi-level buffering architectures with heterogeneous packet values are considered in [25, 26, 27].

Several works have concentrated on adding the *required processing* characteristic, i.e., how many processing cycles the packet has to go through before it can be transmitted, a characteristic motivated by many different tasks of varying complexity that a modern network processor can apply to a packet. The work [7] considered a single queue buffering architecture in various settings. Later, the work [28] introduced the notion of *lazy* algorithms together with a new accounting infrastructure that has allowed to prove tight bounds on the competitiveness of the lazy greedy push-out algorithm that are logarithmic in the maximal required processing k. Lazy algorithms were further considered in [29], where processing order for the packets was decoupled from transmission order, culminating in the LPQ (lazy priority queue) algorithm that was proven to be 2-competitive. Packets with required processing were also considered in the case of multiple separated queues [30] and a shared memory switch with multiple output ports [24, 31]; each of these works introduced new policies and proved constant upper bounds on their competitive ratios.

Apart from upper bounds on competitiveness, another important class of results in competitive analysis are adversarial lower bounds that hold over all algorithms. Such bounds, when they can be proven, indicate that one cannot hope to get an optimal online algorithm, and a clairvoyant offline algorithm will always be able to outperform it. One well-known example of such a bound is the lower bound of $\frac{4}{3}$ on the competitive ratio of any algorithm in the model with multiple queues in a shared memory buffer and uniform packets (i.e., packets with identical value and required processing) [32, 18]. For the case of a single queue, in the FIFO model with variable values and uniform processing there has been a line of adversarial lower bounds culminating in the lower bound of 1.419 that applies to all algorithms [22], with a stronger bound of 1.434 for the special case of $B = 2$ if all possible values are admissible [20, 21]. In the case of variable processing with uniform values, no general lower bounds for FIFO order are known apart from a simple lower bound of $\frac{1}{2}(k + 1)$ for greedy non-push-out policies [33].

In other related directions, policies with memory have been considered in [34, 35, 36], and a special domain-specific language intended to express buffer management policies for user-defined policies has been proposed in [37]. Pruhs [38] provides a comprehensive overview of scheduling for server systems; however, scheduling for server systems usually concentrates on average response, while we focus mostly on throughput.

There are two works [6, 8] that are closely related to this paper and consider weighted throughput (total transmitted value) as an optimized objective. In the model from [6], every packet has an intrinsic value and slack; the required processing (work) per packet is assumed to be identical. The buffer size in this case is bounded implicitly since there is an upper bound on the slack. It turns out that a simple greedy algorithm that transmits the most valuable packet whose slack has not expired yet is at most 2-competitive. The second work [8] considers packets with an intrinsic value and required processing stored in a limited buffer but does not take into account latency constraints (slack); in the special case, where only two values 1 and V are possible and processing requirements per packet vary in a range from 1 to W, the authors propose an $(1 + \frac{W+2}{V})$-competitive algorithm. It remains an open problem to see whether there exist algorithms with constant competitiveness in the general case when both intrinsic values and processing requirements can be arbitrary integers from 1 to V and 1 to W respectively. In this work, we extend these models to consider all three characteristics (intrinsic value, required processing, and slack) at once and show that there exist algorithms with constant competitiveness even in these settings.

3. Model Description and Priority Queues

We use a model similar to the one introduced in [6]. Consider a single queue that handles the arrival of a sequence of unit-sized packets. While previous works [6, 8] have dealt with the case of two characteristics (value and slack in [6], value and required processing in [8]), we assume that each arriving packet p is branded with three characteristics: (1) the number of required processing cycles (*work*) $w(p) \in \{1, \ldots, W\}$, (2) its processing *value* $v(p) \in \{1, \ldots, V\}$, (3) *slack* $s(p) \in \{1, \ldots, S\}$ that defines how long from the arrival time a packet should be transmitted before it is lost without any gain to the final objective (weighted throughtput). We denote by $[\![w \mid v \mid s]\!]$ a packet with work w, value v, and slack s; a sequence of n packets with the same parameters, by $n \times [\![w \mid v \mid s]\!]$. Similar to [6], we assume unbounded queue size; note that the maximal slack value S effectively bounds it. In the notation above, slack s of a packet decreases on every time step, i.e., we are talking about "current", effective slack. A packet is *fully processed* if the processing unit has scheduled the packet for processing for at least its required number of cycles.

Although the maximal work W, maximal value V, and maximal slack S will play a fundamental role in our analysis, the proposed online algorithms do not need to know them in advance. Note that for $W = 1$ the model degenerates into a single queue of unit-sized packets with heterogeneous intrinsic values, as in [6].

We assume discrete slotted time with three phases in each time slot: (i) *arrival*: new packets arrive, and admission control decides if a packet should be dropped or, possibly, an already admitted packet should be pushed out; (ii) *processing*: one packet is selected for processing by the scheduling unit; (iii) *transmission*: at most one fully processed packet is selected for transmission and leaves the queue. If a packet is *dropped* prior to being transmitted (while it still has a positive number of required processing cycles), it is lost. A packet may be dropped either upon arrival or due to a push-out decision while it is stored in the buffer. A packet contributes its value to the objective function only upon being successfully transmitted; note that only one packet may be transmitted per time slot. The goal is to devise buffer management algorithms that maximize the overall weighted throughput, i.e., the total value of all packets transmitted out of the queue.

It is well known that different processing orders have a significant impact on the performance of buffer management policies [7, 8]. For a single queue buffering architecture with bounded buffers, one usually has to consider two orders: *admission order* that defines which packets will be dropped in case of congestion and *processing order* of packets in the queue. In the bounded-delay model [6], a queue has unbounded size, so during admission an algorithm can accept all packets, later automatically discarding packets with expired slack, and we only need to define the processing order. Hence, no algorithm we consider below ever drops a packet upon arrival: PQ algorithms defined in this section push out only when the slack expires (becomes less than required work), and the SPQ algorithm defined in Section 5 has more complicated pushout decisions but also always accepts all arrivals. Note that although the queue is unbounded, it can contain only a limited number of packets at the end of the next arrival phase, defined by the maximal slack value and required processing[3]. In what follows, we introduce several processing orders to understand their impact on weighted throughput (total transmitted value).

Priority queues with push-out are a reasonable choice for packet processing algorithms, and previous research indicates that they often lead to good competitive ratios. In settings with a single characteristic, the corresponding

[3]Our model does not limit a number of packets during arrival phase. In practical implementations to guarantee a limited buffer size at every point in time, we can discard least valuable packets according to processing order, whose slack is bigger than the total work of all currently admitted packets.

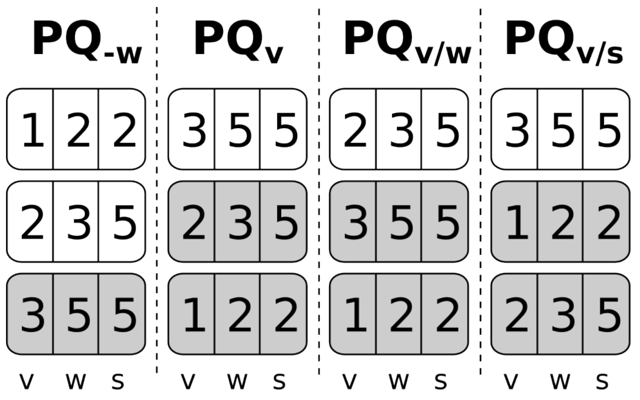


Figure 1: A sample time slot for PQ_{-w}, PQ_{v}, $PQ_{v/w}$, and $\mathrm{PQ}_{v/s}$. The head of line packet is at the top. Shaded packets are those that will not be transmitted due to lack of time.

priority queue is often optimal; for example, the work [7] shows this for packets with varying required processing (and unit value and slack). However, in the case of multiple characteristics priority queues have been recently shown to be non-competitive in the worst case, i.e., with non-constant lower bounds on the competitive ratio [8]. Still, priority queues remain an attractive practical choice, and it is important to study their properties in situations with slack as well; following [8], we introduce the following definition.

Let f be a function of packets, $f(w, v) \in \mathbb{R}$, with the intuition that better packets have larger values of f. Then the PQ_f processing policy (*priority queue*) is defined as the policy that processes packets in its queue in the decreasing order of the values of f on these packets.

In particular, we consider the following specific priority queues; below w denotes the current residual work, v denotes the value, and s denotes slack:

(1) $\mathrm{PQ}_{-w} = \mathrm{PQ}_{-w+v/(V+1)}$ orders packets in the increasing order of their required processing, breaking ties by value;

(2) $\mathrm{PQ}_{v} = \mathrm{PQ}_{v-w/(W+1)}$ orders packets in the decreasing order of their value, breaking ties by required processing;

(3) $\mathrm{PQ}_{v/w}$ orders packets in the decreasing order of their value-to-work ratio, i.e., it prioritizes packets that yield the best value per one time slot of processing; one can break ties either by work or by value, all statements below will remain valid, so we unite these variations;

(4) PQ_{v/w_0} orders packets in the decreasing order of their value-to-original-work ratio, i.e., in difference from $\mathrm{PQ}_{v/w}$ the priority of a packet does not increase as it gets processed more and its work decreases;

(5) $\mathrm{PQ}_{v/s}$ and PQ_{v/s_0} order packets in the decreasing order of their value-to-slack and value-to-original-slack ratio respectively; this idea is known in scheduling, where it has been shown to work well as a variation on the earliest due date (EDD) heuristic [39].

Figure 1 illustrates a sample time slot with four different processing orders; the queues are shown vertically, with head of the queue on top. Shaded packets are those that the corresponding algorithm will not have time to process; this illustrates the differences between four processing orders.

4. When Online Algorithms Fail: Lower Bounds

In this section, we show a number of *lower* bounds for the competitiveness of online algorithms, i.e., results that show how bad a given online algorithm is in the worst case when compared to a clairvoyant offline algorithm. We first concentrate on *general* lower bounds, i.e., bounds that hold for all deterministic online algorithms; note that while proving a lower bound for a specific algorithm usually reduces to simply providing a counterexample, general lower bounds valid for all or a specific wide class of algorithms are usually proven by adversarial arguments and often represent interesting results. In this section, we provide a number of lower bounds in conditions that gradually become more beneficial for the online algorithms.

In Theorem 1, we show that in the general setting introduced above, no deterministic online algorithm can have even a sublinear competitive ratio, let alone a constant one. The worst-case example, however, will rely on "just-in-time" packets, i.e., packets $[\![w \mid v \mid s]\!]$ with $w = s$, which requires the algorithm to either begin processing or drop this packet immediately as it arrives. In subsequent theorems, we relax this setting by imposing additional constraints on slacks. In Theorem 2, we show a general lower bound for the case when all slacks are separated from required work values in an *additive* way, i.e., for every incoming packet $[\![w \mid v \mid s]\!]$ we have $s \geq w + c$ for some constant c. It turns out that this additive separation is insufficient, and we still obtain a non-constant general lower bound.

Our main positive result in the present work is that while additive separation is not enough, *multiplicative* separation is: in Section 5 we present an algorithm that has constant competitiveness under the assumption that every incoming packet $[\![w \mid v \mid s]\!]$ has $s \geq cw$ for some constant c. However, this algorithm will be quite involved, and in this section we also proceed to show that simple priority queues are insufficient even with a multiplicative separation: in Theorem 3 we show a non-constant lower bound for five different priority queues introduced above even under multiplicative separation, Theorem 4 shows the same for arbitrary non-preemptive algorithms, and in Theorem 5 we consider algorithms with α-preemption, a middle ground between preemptive and non-preemptive algorithms that has proven to be useful in other settings.

We begin with a general lower bound without additional constraints; it turns out that in case of three characteristics, there is a relatively simple general lower bound that applies to all online algorithms and is not only non-constant but linear in V.

Theorem 1. *The competitiveness of any deterministic online algorithm A cannot be bounded from above by a constant.*

Proof. We prove the theorem by contradiction. Suppose that an algorithm A has constant competitive ratio; we present the following sequence. At the first time slot, two packets arrive, one with value 1 and slack equal to work equal to 1 and another with value V, and slack equal to work equal to $W = V^2$. The next elements in the sequence are constructed in an adversarial way, depending on A's behaviour. While A is working on the packet with value V, a new packet with value 1, slack 1 and work 1 arrives on every time slot; hence, the optimal algorithm can process all small packets for a total value of V^2 while A has only processed value V. If at any time A chooses any packet other than the large one (with value V and work $W = V^2$), the packet sequence stops immediately, and A's total transmitted value is 1 while the optimal algorithm can finish the larger packet for a total value of V. In both cases, the competitiveness of A is at least V. □

However, the crucial point in this lower bound is that it uses "real-time" packets with $s = w$, so the algorithm is forced to make a decision about them immediately and cannot store smaller packets in its buffer "for later".

Unfortunately, even if we try to separate the slacks from required processing values by some *additive* constant c (i.e., $s(p) \geq w(p) + c$ for all packets), there are no deterministic online algorithms with constant competitiveness.

Theorem 2. *The competitiveness of any deterministic online algorithm* ALG *cannot be bounded from above by a constant even if there is a predefined constant c such that for every packet $s(p) \geq w(p) + c$.*

Proof. We prove the theorem by contradiction. Suppose that an algorithm ALG has constant competitive ratio; we present the following sequence. At the first time slot, two packets arrive, one with value 1, work $2c$ and slack $3c$ and another with value V, work $W = 2cV^2$ and slack $S = 2cV^2 + c$. The next elements in the sequence are constructed in an adversarial way, depending on ALG's behaviour. The adversary waits for the first $2c$ slots and checks whether ALG has processed at least c work of the first packet. If it has not, another packet similar to the first one arrives, and the process is repeated until the large packet is completely processed by ALG. Thus, ALG will process the large packet and at most one small packet, with total value $V + 1$, while the optimal algorithm can process all of the small packets with total value V^2. If ALG has spent at least c time units worth of processing on any small packet then the packet sequence immediately terminates, and as a result ALG has no time to process the large packet. In this case, ALG will have processed at most 2 small packets with total value 2, while the optimal algorithm, having anticipated this in advance, can process the large packet, getting total value V. In both cases, the competitiveness of ALG is at least $\min(\frac{V}{2}, \frac{V^2}{V+1})$. □

However, in our main result below (Theorem 6) we will see that if we require the slack s to be separated from w by some constant multiplicative factor (i.e., $s(p) > cw(p)$ for some constant c and for every packet p), then we will be able to achieve a constant competitive ratio.

But first we show a few more lower bounds for specific algorithms that are interesting from the practical point of view. Lower bounds on competitive ratios represent only individual hard cases, specific sequences of packets where the algorithm loses a lot to the optimal. Nevertheless, lower bounds can still provide important insights to the comparative quality of online algorithms. Note that while previous work has provided non-constant lower bounds for priority queues in various settings with two characteristics [8], those were proven for a bounded buffer and hence are inapplicable in our case.

Theorem 2 already shows that if "real-time" packets are allowed no online algorithm is competitive. Still, it turns out that baseline priority queues are non-competitive even with large slacks (by large slack we will denote a slack which is always greater than the work by some constant c).

Theorem 3. *If w, s, and v are admissible values of work, slack, and value respectively, and s is always greater than cw for some constant c, the competitiveness of priority queue algorithms PQ_v, $\mathrm{PQ}_{v/w}$, PQ_{v/w_0}, $\mathrm{PQ}_{s/w}$, PQ_{s/w_0} is at least*

$$\frac{1-(\frac{w-2}{w})^{\frac{1}{w}\left\lfloor \log_{\frac{w}{w-2}} v\right\rfloor}}{1-\frac{w-2}{w}}\frac{1}{c},$$

which is non-constant in v and w.

Proof. Consider a sequence of n packets arriving one per timeslot. All packets have the same work w and slack $s = cw$, and their values are respectively v, $v\frac{w}{w-2}$, $v\left(\frac{w}{w-2}\right)^2$ and so on, with the ith packet's value $v(\frac{w}{w-2})^i$; since $v \geq 1$ n is limited by $n \leq \log_{\frac{w}{w-2}} v$. At time i, the priority queue algorithm will always prefer the $(i+1)$th packet since all parameters in question (v, w, v/w, v/w_0, v/s, and v/s_0) are better for the new packet. Thus, the first fully processed packet will be the last packet, and the total number of processed packets will not exceed c, so the total processed value can be bounded from above by $v(\frac{w}{w-2})^n c$. On the other hand, the optimal algorithm can process the nth, $(n-w)$th, $(n-2w)$th packets, and so on, with total value $v(\frac{w}{w-2})^n\frac{1-(\frac{w-2}{w})^{\frac{n}{w}}}{1-\frac{w-2}{w}}$. This yields the necessary bound, and it tends to infinity if v and w tend to infinity. $\square$

Theorem 3 shows a common problem of priority queue algorithms: due to preemptiveness they abandon packets they have already invested in, and thus sometimes a large part of the processing time is wasted.

On the other hand, if we consider non-preemptive algorithms the situation is even worse.

Theorem 4. *If w, s, and v are admissible values of work, slack, and value respectively, and s is always greater than cw, then if $w > c$ the competitiveness of any non-preemptive algorithm is at least V.*

Proof. Again we prove by contradiction, constructing an adversarial input sequence for an algorithm A that is supposed to have constant competitive ratio. At the first time slot, a single packet with value 1, work w, and slack cw arrives. If A never begins to process it will have infinite competitive ratio. If at some time moment it starts processing the first packet then a new packet with value V, work 1, and slack c is sent, and A will have no time to process it (since it finishes processing the first packet after the slack of the second one expires), losing V times to the optimal algorithm. $\square$

Thus, neither preemptive nor non-preemptive priority queue algorithms can be competitive even with a multiplicative gap between required processing values and slacks. They have different problems, though: preemptive algorithms often work in vain, on packets that will later be pushed out, while non-preemptive algorithms never do that but often work on packets with little "bang for the buck", too small value as compared to their required processing. There is nothing we can do about the latter problem if preemption is impossible, so the solution must be to limit preemption in some way to alleviate the former problem.

One possible approach to get the best of both worlds is to introduce a property that would be in between preemptive and non-preemptive algorithms; we call this property *α-preemption*. Consider a priority queue algorithm which sorts packets by some function f. We make it more stable in the following way: if no packet is processed at the moment, just choose a packet p with the largest $f(p)$ as usual, but if the algorithm is currently processing some packet p then it will switch to another packet p' only if $f(p') \geq \alpha f(p)$. For $\alpha > 1$, this introduces a commitment to already processed packets, helping to solve the above problem, and we will see them in experimental evaluation. The idea of preempting only if the newly arriving packet (job) is better by some predefined factor has appeared before in the context of job scheduling; see, e.g., [40].

However, in the worst case α-preemption also does not really help; we illustrate it below for $\mathrm{PQ}_{v/w}$, and similar examples can be constructed for other priority queues.

Theorem 5. *The $\mathrm{PQ}_{v/w}$ algorithm with α-preemption has a lower bound on its competitiveness of $\Omega(w)$ for arbitrarily large slacks.*

Proof. Consider an input sequence that consists of two kinds of packets. There are k "large" packets with work w, value v, and slack cw each; c "large" packets arrive on the first time slot, and the others arrive at times $\frac{w}{2}$, $\frac{2w}{2}$, $\frac{3w}{2}$,$\frac{4w}{2}$,$\cdots$, $\frac{w}{2}(k-c)$. There also are $k-c+1$ "small" packets with work 1, value $\frac{\alpha+1}{w}v$, slack c, and arrival times $\frac{w}{2}+c-1$, $\frac{2w}{2}+c-1$, $\ldots$, $\frac{kw}{2}$. $\mathrm{PQ}_{v/w}$ will process all small packets and c large packets for a total value $cv+\frac{(k-c+1)v(\alpha+1)}{w}$ while another algorithm could process only large packets for a total value of kv which can be greater than $cv+\frac{(k-c+1)v(\alpha+1)}{w}$ by an arbitrarily large factor if w is sufficiently large. Here we assume that $\mathrm{PQ}_{v/w}$ chooses the oldest package among the packages with the same v/w. If this is not the case we can add k to the value of the first "large" package, $k-1$ to the value of the second and so on. $\square$

While algorithms with α-preemption solve some problems of priority queue algorithms, two major problems remain. The first problem is that even with large slacks the algorithm can drive itself into a "real-time" situation by processing packets with expiring slack (we have pressed upon this weakness in Theorem 5). The second problem is that while preemption does lend some stability and robustness to the algorithm, once an algorithm has abandoned processing of some packet it is not going to prefer it to unprocessed packets, wasting invested time.

How does one solve these problems? For the former problem, the algorithm could introduce a special extra "reserve time" so that packets that could force it into a real-time situation would simply not be accepted. For the latter, it could introduce additional restrictions on preemption that would favor packets that the algorithm had already worked on in the past. In the next section, we present an algorithm that implements these ideas and consequently is free of these problems; we also identify conditions when it has a constant competitive ratio.

5. Constant Competitiveness under Multiplicative Separation between Slacks and Required Work

We have seen in Theorem 1 that any deterministic online algorithm is non-competitive in the general case, and in Theorem 2 extended this result to additive separation between the slack and required work of a packet. Surprisingly, as soon as we require any, however small *multiplicative* separation between w and s, we can design an algorithm with constant competitiveness!

The main result of our work is a new upper bound for the model with three characteristics. We begin by describing the algorithm and basic assumptions and then proceed to prove the bound. For the proof, we introduce the following *assumption*: suppose that for every packet $[\![w \mid v \mid s]\!]$, its initial slack at time of arrival is at least c times larger than required work, $s \geq cw$. The proof will work for any constant $c > 1$, and the algorithm does not have to know c in advance.

The main idea of the algorithm is to enhance the α-preemptive $\mathrm{PQ}_{v/w}$ algorithm with two new ideas designed to solve the problems highlighted in Theorem 5:

- special priority assignment rules that reduce the value of new packets and/or increase the value of packets that have already been partially processed (to avoid wasted effort due to too much preemption);
- an additional rule (we call it β-pessimism) which restricts the algorithm from working with packets with expiring deadlines, so it can never corner itself in "just-in-time" situations.

The resulting algorithm can be thought of as a priority queue with two new modifications:

Algorithm 1 PQ BUFFER MANAGEMENT

```
1: for every packet $p$ arrived at the current time slot do
2:     ARRIVAL($p$)
3: end for
4: $p \leftarrow$ packet with maximal priority
5: PROCESS($p$)
6: for every p in the buffer do
7:     CHECKPUSHOUT($p$)
8: end for
```

Algorithm 2 ARRIVAL(p=$[\![w \mid v \mid s]\!]$)

```
1: Priority$(p) \leftarrow \frac{v(p)}{\alpha \cdot w(p)}$
2: Processing$(p) \leftarrow false$
3: Add $p$ to the buffer
```

- every packet has a flag that shows whether the algorithm has already begun processing it;
- if a packet has not begun processing yet (it is a "new" packet), then for the purposes of preemption (i.e., the order of the priority queue) its value is reduced by a factor of α (this is the α-preemption property);
- moreover, in order for new packets to begin processing they need to have an extra "cushion" on the slack: the algorithm makes a provision to spend share β of the new packet's processing time on other packets due to preemption, so work on the new packet can begin only if its slack is at least $(1+\beta)w$ (we call this property *β-pessimism*).

The new algorithm is simply a priority queue $PQ_{v/w}$ with the two modifications above. We have presented this algorithm in pseudocode Algorithm 1, with specific procedures outlined in Algorithms 2–4. Algorithm 3 shows how in the first processing cycle we set the flag $\text{Processing}(p) = \text{True}$ and increase priority by a factor of α, thus making it α times harder for new packets to push out packets that the algorithm has already worked on. Note that after the algorithm has started processing a packet p neither the $\text{Processing}(p)$ flag nor $\text{Priority}(p)$ will ever change. Thus, all packets with $\text{Processing}(p) = \text{True}$ are processed in the LIFO order.

In this section, we work with one possible formal implementation of this idea based on organizing "already touched" packets (packets with $\text{Processing}(p) = \text{True}$) in a stack. This algorithm is equivalent to Algorithm 1, but a stack-based representation will be useful for us to prove the upper bound on competitiveness.

Definition 5.1. *The stack-based α-preemptive, β-pessimistic $PQ_{v/w}$ algorithm, or* SPQ *(stack with priority queue) operates as follows: apart from the queue, it emulates a stack and on every step processes the top packet in the stack, i.e., some packets are "on stack" and some are "not on stack" in the buffer.* SPQ *always processes the top packet on the stack*[4]*, denoted* HOL *with parameters $[\![w_h \mid v_h \mid s_h]\!]$; we also denote by $\text{HOL}^t = [\![w_h^t \mid v_h^t \mid s_h^t]\!]$ the* HOL *packet at time t. By $w(t)$ and $s(t)$ we will denote remaining work and slack of a packet at the beginining of time slot t, and we will refer to the initial work and slack by ommiting the parameter t.* SPQ *operates as follows: on every time slot,*

(R1) drop all packets $[\![w \mid v \mid s]\!]$ from the buffer that are not on stack and have slack

$$s(t) < (1+\beta)w(t) = (1+\beta)w;$$

*this is called the β-*pessimism rule*: we never put on stack a packet without some cushion on its deadline; but if it is already on stack it will remain there;*

(R2) for each new packet, we accept it on stack (making it the new HOL*) if its $\frac{v}{w}$ ratio is at least α times larger than the current* HOL*'s (the α-*preemptiveness*);*

(R3) drop packets from the stack with insufficient time to process ($[\![w \mid v \mid s]\!]$ with $s(t) < w(t)$).

Algorithm 3 PROCESS(p=$[\![w \mid v \mid s]\!]$)

```
1: if not $\mathrm{Processing}(p)$ then
2:     $\mathrm{Processing}(p) \leftarrow \mathrm{True}$
3:     $\mathrm{Priority}(p) \leftarrow \alpha \cdot \mathrm{Priority}(p)$
4: end if
5: Spend one processing cycle on packet $p$
```

Algorithm 4 CHECKPUSHOUT(p=$[\![w \mid v \mid s]\!]$)

```
1: if $w(p) > s(p)$ then
2:     drop packet $p$
3: end if
   ▷ check slack against expected work for unprocessed packets
4: if (not $\mathrm{Processing}(p)$) and $w(p) \cdot (1+\beta) > s(p)$ then
5:     drop $p$
6: end if
```

The basic idea of the stack in SPQ is to memorize which packets we have already worked at and which we have not. They are then treated differently: if SPQ has already worked on a packet, it does not need the α factor to preempt for it; if it is a new packet, it has to be α times better to get to the stack. Note that the complexity of the SPQ algorithm does not differ from the complexity of a priority queue: on every timeslot, SPQ needs to update a priority queue and spend $O(1)$ operations to update the stack. We assume that a hardware implementation of a priority queue is available for PQ and SPQ implementation in a real network element.

Note that SPQ has two parameters, α and β. Figure 2 illustrates SPQ with a few sample time slots for $\alpha = 2$ and $\beta = 1$; shaded packets are being dropped (due to the pessimistic rule), the dotted rectangle surrounds SPQ buffer divided into the stack and "other" packets, and grey dashed arrows show how packets move in SPQ buffer and are transmitted out. At $t = 1$, a new packet has $\frac{v}{w} = 2$ which is more than $\alpha = 2$ times larger than the HOL packet's $\frac{v}{w} = \frac{1}{2}$, so it is accepted on top of the stack and processed at $t = 2$. Then the previous HOL returns to top of the stack and is processed; meanwhile, the $[\![3 \mid 2 \mid 5]\!]$ packet residing in the buffer has been reduced to $[\![3 \mid 2 \mid 3]\!]$ and dropped due to the pessimistic rule; a new packet is accepted on stack when the stack becomes empty. Note that in this short sequence, SPQ operates suboptimally: it would be better not to drop $[\![3 \mid 2 \mid 3]\!]$ and accept it on stack instead.

Note also that due to the β-pessimism rule the SPQ algorithm can be sometimes strictly improved: in some cases, SPQ pushes out all packets and sits idle with an empty buffer even when packets with $w(p) \cdot (1+\beta) \geq s(p) \geq w(p)$ could be available for processing. However, push-out rules can be modified so that the algorithm will not have this undesirable property, while at the same time still having the same upper bound on competitiveness. Informally, if there are no packets to work on according to the SPQ algorithm we can work on whatever packet SPQ would want to drop but whose slack has not expired yet. Algorithm 5 shows a version of the CHECKPUSHOUT procedure modified according to this idea; in the improved version, a packet p is pushed out only if $w(p) > s(p)$, but if $w(p) \cdot (1+\beta) > s(p)$ its priority is set to zero, so it will not be processed unless there are no more packets with nonzero priorities.

Theorem 6. *If $s \geq cw$ for every packet at time of arrival, the* SPQ *algorithm defined above has competitive ratio at most*

$$\left(1 + \frac{1}{\beta(\alpha-1)-1}\right)\left(1 + \alpha + \frac{\alpha c}{(c-\beta-1)}\right).$$

Proof. The first (crucial) step of the proof is to pass from computing SPQ value as a sum of the values of transmitted packets to computing it as a sum of (perhaps partially) processed packets. This is an important idea that makes the rest of the proof much easier and may be later reused for other results. To do so, we introduce *bites*. A *bite* is the "value" of a single processing timeslot, i.e., $\frac{v_h}{w_h}$ for HOL $[\![w_h \mid v_h \mid s_h]\!]$.

Formally, the transition to bites is summarized in the following lemma. In essence, the statement of this crucial lemma says that due to the two modification we explained at the beginning of this section, the SPQ algorithm very

[4]This means that SPQ can accept a new packet on top of the stack, interrupting the current, and then return back to it when the new packet is transmitted.

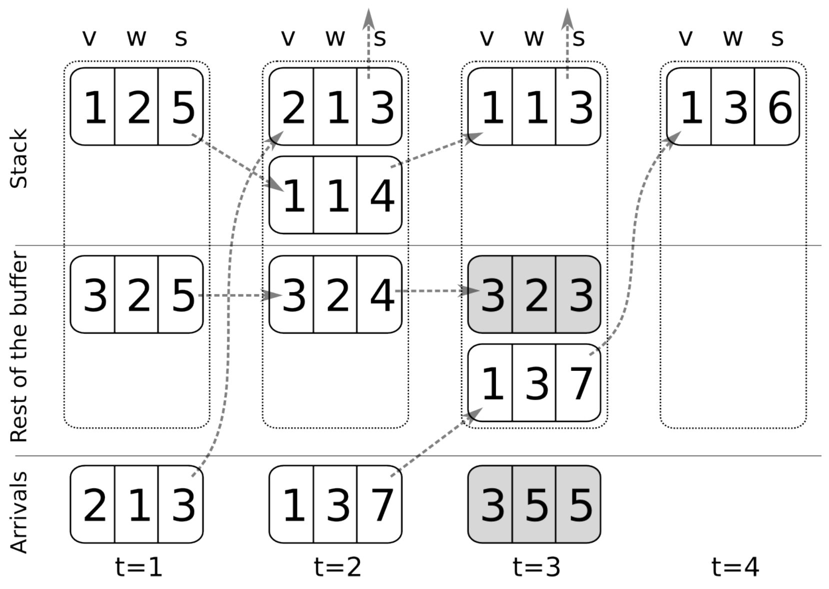


Figure 2: Sample operation of the SPQ algorithm.

Algorithm 5 CHECKPUSHOUT2(p=$[\![w \mid v \mid s]\!]$)

```
1: if $w(p) > s(p)$ then
2:     drop packet $p$
3: end if
   ▷ check slack against expected work for unprocessed packets
4: if not Processing and $w(p) \cdot (1+\beta) > s(p)$ then
5:     Priority$(p) \leftarrow 0$
6: end if
```

seldom spends working cycles on packets that will never be finished due to preemption; the "very seldom" part is formalized below as a multiplicative constant. Therefore, after adding a proper multiplicative discount to the resulting value we can move from counting completed packets to counting the time spent on processing them.

Lemma 7. *There exists a partition of total value $\mathcal{V}$ obtained by* SPQ *over its entire operation among all timeslots $g : [1, T] \to \mathbb{N}$, $\sum_t g(t) = \mathcal{V}$, such that on every timeslot t, $g(t) \geq \left(1 - \frac{1}{(\alpha-1)\beta}\right) \frac{v_h^t}{w_h^t}$.*

Proof. The idea is to take the value of packets actually processed (transmitted) by SPQ and distribute it among "fruitless" time slots, when SPQ was processing a packet only to drop it afterwards.

What does it mean that a packet $[\![w \mid v \mid s]\!]$ has been dropped from the stack? By (R1), when it was accepted it had a slack of at least $(1+\beta)w$ timeslots. Since it has not finished processing, it means that for at least βw timeslots SPQ has been processing a better packet that was accepted on top of stack by (R2). This packet has at least α times better v/w ratio.

Naturally, it might happen that a packet was preempted by a better packet, and that one was in turn preempted by another, and so on. However, each new packet must be α times better than the previous one. Therefore, for every timeslot when we processed a packet we can redistribute $\frac{1}{\alpha}$ of its value to a preempted one, $\frac{1}{\alpha^2}$ to a previously preempted packet, and so on, so as a result the total redistributed value is bounded from above by

$$\frac{1}{\alpha} + \frac{1}{\alpha^2} + \frac{1}{\alpha^3} + \ldots = \frac{1}{\alpha - 1}.$$

Now, for every processed packet we need to redistribute out βw of its processing timeslots to previously preempted packets, so the value of each processed timeslot is reduced by a factor of at least $1 - \frac{1}{(\alpha-1)\beta}$. □

We now proceed to proving the theorem. From this moment on, we will consider bites instead of full packets, and by Lemma 7 we can simply multiply the resulting upper bound by δ, where

$$\delta = \frac{1}{1 - \frac{1}{(\alpha-1)\beta}} = \frac{\beta(\alpha-1)}{\beta\alpha - \beta - 1}.$$

Let us now classify all bites that OPT has processed over time, creating a matching of bites between OPT and SPQ. For a bite out of a packet $p = [\![w \mid v \mid s]\!]$, there are three cases:

(1) this bite has also been processed by SPQ;

(2) this bite has not been processed by SPQ, and either:

  (a) at the time when OPT was processing it p has not been accepted on stack by (R2), or

  (b) at the time OPT was processing it p was on stack;

(3) this bite has not been processed by SPQ because p:

  (a) has not been accepted on stack by (R1);

  (b) has been removed from stack by (R3).

In general, different bites of the same packet p could fall into different cases. If this is the case, we split a packet into parts p_1, p_2 and p_3 and in further analysis view them as separate packets (this lets us merge cases 3a and 3b into one).

Note that all of these three cases may cover the same packet processed by SPQ independently, so the resulting ratio will be the maximum of ratios from cases 1–3.

In case 1, we match the bites one-to-one, and they are covered by Lemma 7, with a competitive ratio of δ. In case 2, we match OPT's bite to a corresponding bite of the packet $p' = [\![w' \mid v' \mid s']\!]$ that SPQ processed on the same time slot. By (R2), p' has the value-work ratio of at most α times worse than p: $\frac{v'}{w'} > \frac{1}{\alpha}\frac{v}{w}$. Therefore, the matching bite brings SPQ at least α times less than the bite of OPT, and by Lemma 7 we get a competitive ratio of $\alpha\delta$.

Case 3 is the most interesting. To bound the competitive ratio in case 3, we denote by t^p_{arr} the time when packet p arrives, by t^p_{beg} the time when OPT begins processing p, and by t^p_{end} the time when OPT finishes processing p. We sort the packets in the order of t_{end} and define a mapping between arriving packets and SPQ bites.

Definition 5.2. *Consider a stream of packets $(p_1, p_2, \ldots)$ sorted by t^p_{end}. The* arbitrary packet mapping *is defined as follows: for each $p = [\![w \mid v \mid s]\!]$, map it to an arbitrary subset of $\frac{1}{\alpha}w$ unmapped* SPQ *bites from the interval $[t^p_{\text{arr}}, t^p_{\text{beg}}]$.*

Lemma 8. *The arbitrary packet mapping from Definition 5.2 is feasible, and therefore each packet processed by* OPT *and dropped by* SPQ *according to (R1) has at least $\frac{c-1-\beta}{c}$ of its bites mapped to processed* SPQ *bites.*

Proof. We denote for convenience $\gamma = c - 1 - \beta$, so it always holds that for every packet $[\![v \mid w \mid s]\!]$, at time of arrival $(\gamma + 1 + \beta)w \le s$. We have to prove that for every packet $p = [\![w \mid v \mid s]\!]$, there are at least $\frac{\gamma}{c}w$ bites of SPQ left unmapped in the interval $[t^p_{\text{arr}}, t^p_{\text{end}}]$. First, there are at least $s - (1+\beta)w \ge \gamma w$ SPQ bites in the interval in total, since otherwise we would have time to accept p on stack and process it.

Second, note that the already mapped bites from the interval $[t^p_{\text{arr}}, t^p_{\text{beg}}]$ can only be mapped to packets fully processed by OPT over the interval $[t^p_{\text{arr}}, t^p_{\text{end}}]$: if a packet p' has $t^{p'}_{\text{end}} > t^p_{\text{end}}$, it comes later in the ordering, and if it has $t^{p'}_{\text{beg}} < t^p_{\text{arr}}$ then all bites mapped to p' have time smaller than t^p_{arr}. There are at most $s - w$ OPT bites spent on these packets, since $t^p_{\text{end}} - t^p_{\text{arr}} = s$, and OPT needs w timeslots to actually process p; hence at most $\frac{\gamma}{c}(s-w)$ SPQ bites from $[t^p_{\text{arr}}, t^p_{\text{end}}]$ will be already mapped by this step.

Combining the two, we get that in total we have at least γw SPQ bites, and we have already mapped at most $\frac{\gamma}{c}(s-w)$ of them, so this leaves us $\gamma w - \frac{\gamma}{c}(s-w) \ge \gamma w \left(1 - \frac{1}{c}(c-1)\right) \ge \frac{\gamma}{c}w$ bites free to be mapped to p, which proves that the mapping is always feasible. ☐

By Lemma 8, we get that each packet processed by OPT and rejected by SPQ due to (R1) maps to at least $\frac{c}{c-1-\beta}w$ bites of SPQ; since, again, these bites might be later preempted as in case 2, the total competitive ratio in this case is $\frac{c-1-\beta}{c}\alpha\delta$. Therefore, we have obtained a total competitive ratio of

$$\delta + \alpha\delta + \alpha\delta\frac{c}{c-1-\beta} = \frac{\beta(\alpha-1)}{\beta\alpha-\beta-1}\left(1+\alpha+\alpha\frac{c}{c-1-\beta}\right) = \left(1+\frac{1}{\beta(\alpha-1)-1}\right)\left(1+\alpha+\frac{\alpha c}{(c-\beta-1)}\right).$$

□

We are now free to choose α and β to optimize this competitive ratio under natural constraints $\alpha > 1$, $\beta > 0$, $\beta(\alpha-1)-1 > 0$, $\beta < c-1$ (otherwise the algorithm will not be well defined). While it is hard to get an analytic optimum for every specific value of c, the asymptotic result is clear: for large c we can take, e.g., $\beta = \sqrt{c}$, $\alpha = 1+\frac{1}{\sqrt[4]{c}}$, and the competitive ratio will tend to 3 as $c \to \infty$.

6. Special cases

In the previous section, we have presented the SPQ algorithm and proved a constant upper bound on its competitive ratio under a multiplicative assumption on the slacks. In this section, we show two theorems that complement our main result for important special cases where one of the characteristics is bounded from above. It turns out that an equivalent setting with bounded values has already been successfully considered in the work [41], which presents an algorithm D^{over} with constant competitive ratio.

Theorem 9. *If V is bounded from above by a constant, then there exist deterministic algorithms with constant competitiveness with no additional constraints on the slack.*

Proof. We refer to [41, Theorem 5.13] for the proof. □

We also consider the complementary special case when V varies but W is bounded from above by a constant. It turns out that in this case we can find a simpler algorithm, namely PQ_v, which has constant competitiveness even if there is no multiplicative bound on the extra slack.

Theorem 10. *If W is bounded from above by a constant, then there exist deterministic algorithms with constant competitiveness with no additional constraints on the slack.*

Proof. Consider the PQ_v algorithm with 2-preemption; i.e., we only preempt the current HOL packet if the incoming packet has value at least twice higher. To prove constant competitiveness, in this case we can consider a simplified modification of the argument from Lemma 7. For simplicity we assume that pushed out packets are lost forever, and packets that OPT was processing while PQ_v was processing a different packet are also lost for PQ_v forever. Still, there are at most W packets lost for every packet processed by PQ_v. Similar to Lemma 7, for every processed packet p we redistribute half of its value, $\frac{v}{2}$, to the preempted packets: $\frac{v}{4}$ to the packet it has preempted, $\frac{v}{8}$ to the packet preempted by the preempted one, then $\frac{v}{16}$ and so on. As a result, we see that PQ_v is at most $4W$-competitive, which does not depend on V and is a constant if W is bounded from above by a constant. □

7. Evaluation

To evaluate our policies, we have used real traffic traces from CAIDA (Center for Applied Internet Data Analysis) [11]. Unfortunately, such traces provide no information about time-scale, and specifically, how long should a time-slot last that is an internal property of processing network element. This information is essential in our model since the size of a time slot determines traffic burstiness even for fixed packet traces.

The traces provide traffic distributions for incoming packets: we specify the size of a time slot and consider all packets in this window as having arrived at the same time slot. Unfortunately, we are not aware of any publicly available datasets with specified required processing, values, and deadlines/slacks for the packets, so for these parameters we had to rely on random generation.[5]

[5]We have made the source code for our simulations available on `https://github.com/adavydow/spq-sim` naturally, the repository does not contain CAIDA traces.

The simulations, aligned to the discussion above, have the following parameters: V is the maximal value, W is the maximal work, S is the maximal slack, and T is the size of time slot (in seconds) for the CAIDA traces. The size of a time slot has a direct linear relationship with the unit intensity of the incoming stream of packets λ: a longer time slot means that more packets on average arrive on a single time slot. The plots show the share of total value of all arriving packets that has been successfully transmitted as a function of various characteristics. Other characteristics have been sampled uniformly at random from their respective intervals. We have run all simulations up to $1{,}000{,}000$ generated packets in order to obtain a clear and robust evaluation.

We have run the evaluations with several different variations of priority queues and the SPQ algorithm with different parameters (see also Section 3 that introduces priority queues and Section 5 that defines SPQ); namely, we compare:

- PQ_w that orders packets in increasing order of their required processing;
- PQ_v that orders packets in decreasing order of their intrinsic value;
- $\mathrm{PQ}_{v/w}$ that orders packets according to their value-to-work ratios, i.e., value per one processing time slot;
- $\mathrm{NPQ}_{v/w}$, a non-preemptive variation of $\mathrm{PQ}_{v/w}$ that we have also found to be a worthy competitor;
- PQ_s that orders packets according to their slacks, prioritizing packets that have to begin processing most urgently;
- $\mathrm{PQ}_{v/s}$ that uses the value-to-slack ratio to order packets;
- $\mathrm{SPQ}_{2,3.7}$, i.e., SPQ with $\alpha = 2$ and $\beta = 3.7$, the theoretically optimal parameters for the setting on Fig. 3a-c according to Theorem 6;
- $\mathrm{SPQ}_{1.2,26}$, i.e., SPQ with $\alpha = 1.2$ and $\beta = 26$, the theoretically optimal parameters for the setting on Fig. 3d according to Theorem 6;
- $\mathrm{SPQ}_{2.15,0}$, i.e., SPQ with $\alpha = 2.15$ and $\beta = 0$, parameters that were experimentally found to be optimal on Fig. 3a-c.

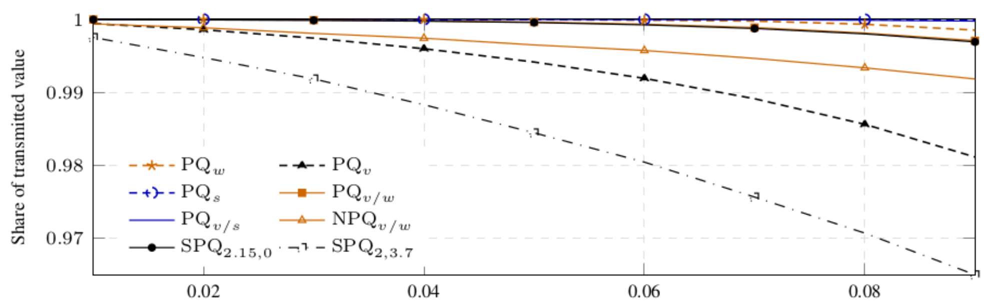
For small congestion levels (Fig. 3a), PQ_s and $\mathrm{PQ}_{v/s}$ are the leaders because these are the algorithms that are last to lose their first packets: if it is possible not to lose a packet, PQ_s will not, so it is best in situations when congestion is low and this possibility has not yet been lost. The same statement does not hold for $\mathrm{PQ}_{v/s}$ but it appears to be a better algorithm overall: we see that for low congestion it is virtually indistinguishable from PQ_s, while for medium and high congestion (Fig. 3b-c) it is noticeably better. We ran SPQ with three sets of parameters, with different theoretically optimal parameter combinations in Fig. 3a-c and Fig. 3d respectively; we see that SPQ performs on par with the best priority queues.

For medium and high congestion levels (Fig. 3b-c), $\mathrm{PQ}_{v/w}$ and $\mathrm{NPQ}_{v/w}$ are the best. An interesting and unexpected observation here is that $\mathrm{NPQ}_{v/w}$ works virtually the same as $\mathrm{PQ}_{v/w}$, the lack of preemption is not a hindrance at all, and for large values of the slack multiplier c (Fig. 3d) $\mathrm{NPQ}_{v/w}$ actually outperforms $\mathrm{PQ}_{v/w}$. For large c, counterexamples such as the ones shown in Theorem 4 cannot appear: in the hard examples, NPQ works on a "bad" packet while good packets arrive and are lost, but for large c it is fine to finish working on a bad packet and only then get to the good ones. At the same time, non-preemptive algorithms tend to *finish* more packets than preemptive ones, which gives $\mathrm{NPQ}_{v/w}$ even a (very) slight advantage.

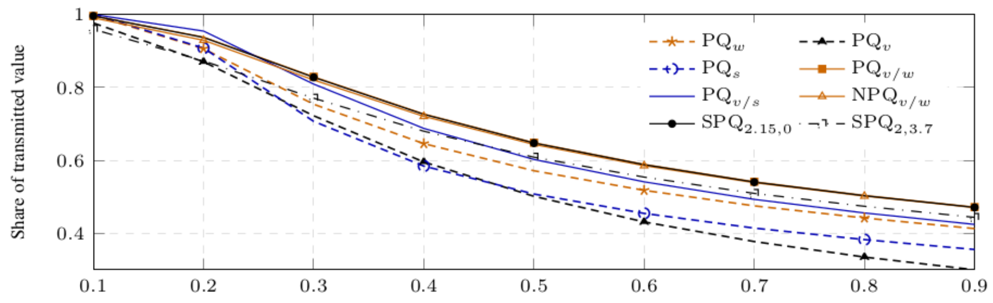
As for the small effects (indistinguishable on the plots), for small c $\mathrm{PQ}_{v/w}$ wins over both $\mathrm{NPQ}_{v/w}$ and SPQ, for large c $\mathrm{NPQ}_{v/w}$ is the best, but there is a region where SPQ (with practically optimized parameters) wins over both.

This happens for an intermediate region of values where c is sufficiently large, but the minimal slack is comparable to the maximal work (so the value of W is sufficiently large too). This region of values is not necessarily practical, but it is interesting to note that SPQ has been shown to not only provide purely theoretical guarantees but also to outperform other algorithms experimentally.

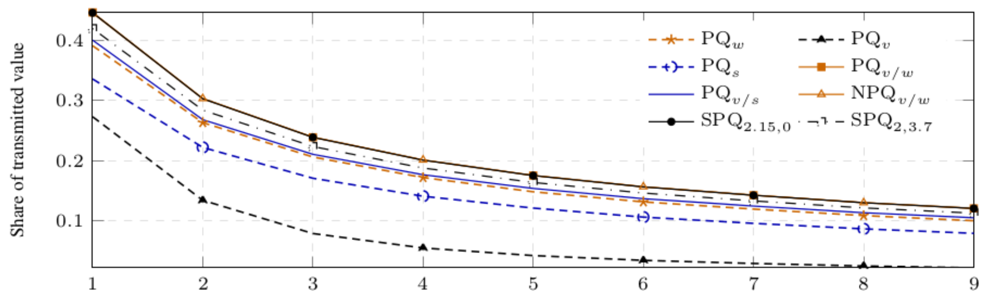
Note that SPQ for large α degenerates to $\mathrm{NPQ}_{v/w}$, but SPQ for small α is not equivalent to $\mathrm{PQ}_{v/w}$ since SPQ operates with original unit values rather than current unit values which PQ recomputes online (this does not matter for NPQ, obviously).



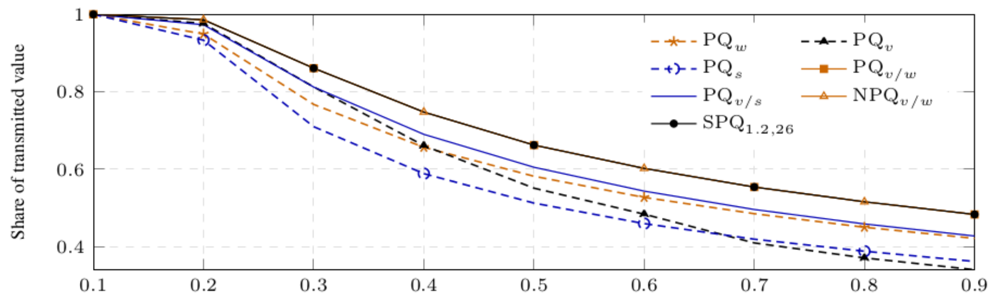
(a) Low congestion: $W = 10$, $V = 10$, $c_{\max} = 10$, $c_{\min} = 5$ T $\cdot 10^{-2}$



(b) Medium congestion: $W = 10$, $V = 10$, $c_{\max} = 10$, $c_{\min} = 5$ T



(c) High congestion: $W = 10$, $V = 10$, $c_{\max} = 10$, $c_{\min} = 5$ T



(d) Large c: $W = 10$, $V = 10$, $c_{\max} = 1000$, $c_{\min} = 100$ T

Figure 3: Share of transmitted value as a function of time slot size T for $W = 10$, $V = 10$.

Figure 4 shows some sample plots for the performance of SPQ with different parameters. It shows that, as expected, for all as α grows the optimal β quickly drops to zero, but Fig. 4a for small c and α the optimal value of β can be nonzero, so this is still a useful parameter and $\beta = 0$ is not always optimal.

In general, our experiments support the theoretical results shown in previous sections and indicate that SPQ may have practical value in addition to the theoretical results. In particular, in this section we have experimentally justified

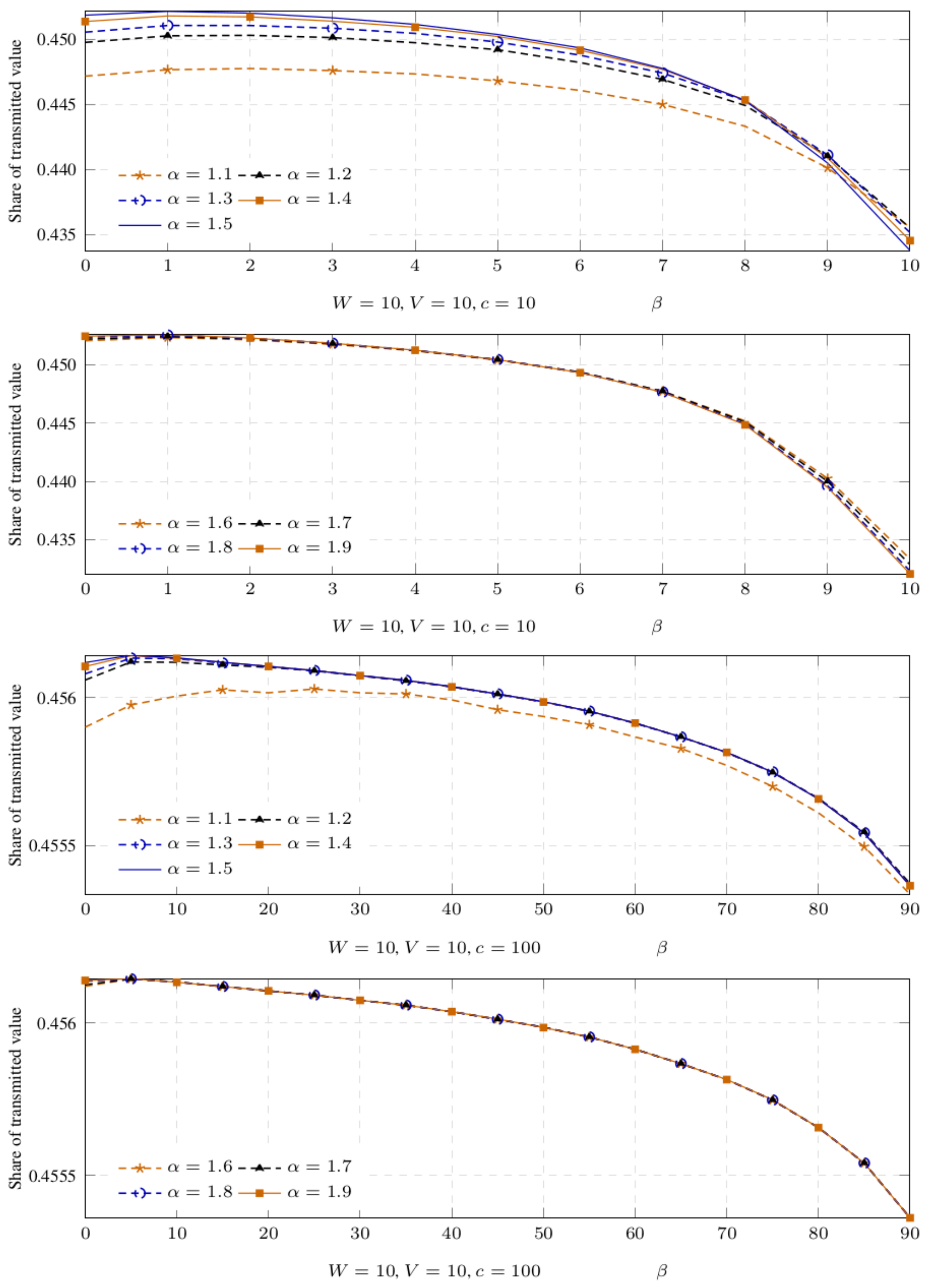


Figure 4: Share of transmitted value as a function of β for different values of α.

the β parameter which on the surface appears to be rather "ad hoc" in the SPQ algorithm.

As for how the proposed model can be deployed in real deployments, it is generally accepted that in order to achieve significant performance gains in congestion control mechanisms used in datacenter environments the end-host control program should be complemented with an additional support in individual switches, at least for delay-sensitive requirements [4, 42, 43, 44, 45]. At this point it is unclear if the packets inside a switch should be scheduled according to some virtual values, SRPT, slacks, or other priorities. In the current paper we shed more light in this direction. In particular, we suggest to represent delay requirements with slacks; the values and required processing allow to multiplex all traffic types in a single weighted throughput objective.

The actual implementation of each of the considered policies would require a corresponding level of flexibility from the network devices. Most importantly, switches must support priority queues, and moreover, these queues must be programmable to an extent which is specific for a given buffer management algorithm. Specifically, PQ_v and PQ_s rely on essentially static priorities, and priorities in $\mathrm{PQ}_{v/s}$ change linearly with time. The most complex priority queues are $\mathrm{PQ}_{v/w}$, $\mathrm{NPQ}_{v/w}$, and PQ_w since their priority values depend on the processing time already spent on each packet. Finally, the SPQ family is almost static—it uses a single priority adjustment on first processing—which makes it actually easier to implement in some existing hardware. Priority-based packet scheduling algorithms are considered to be generally feasible (see, e.g., [4]); recent programming abstractions such as *OpenQueue* [46] or Eiffel [47] make it easier to declare buffer management policies, including the algorithms proposed in this work. In addition, if the need arises to keep track of the slack values across several devices, these devices should either have closely synchronized clocks, e.g., using PTP [48], or decrease the remaining slack on every transmission by leveraging, again, programmable abstractions.

8. Conclusion

In this work, we have considered a very general packet processing model where packets are endowed with three different characteristics: processing requirement (work), intrinsic value (objective function being to maximize total transmitted value), and slack (how much time the algorithm has to process a packet). We have shown that several natural candidates given by various priority queues fail to achieve constant competitiveness. However, we have designed a novel algorithm that operates by emulating a stack with its priority queue and have shown that it has constant competitive ratio which tends to 3 as the slack-to-work ratio increases. On the practical side, we have performed a comprehensive evaluation study on CAIDA network traces [11] that has shown which heuristics work best in this setting. Our results show that even in the seemingly general model one can devise algorithms with good worst-case guarantees. This opens new possibilities for buffer management in latency-sensitive applications with multiple packet characteristics. The results also support the view that frames latencies as constraints to be satisfied rather than a function to be optimized, as does the average flow completion time model.

References

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, Hedera: Dynamic flow scheduling for data center networks, in: USENIX, 2010, pp. 281–296.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. Gruber, Bigtable: A distributed storage system for structured data, in: OSDI, 2006, pp. 205–218.

[3] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, Operating Systems Review 44 (2) (2010) 35–40.

[4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, S. Shenker, pfabric: minimal near-optimal datacenter transport, in: SIGCOMM, 2013, pp. 435–446.

[5] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, S. Shenker, phost: Distributed near-optimal datacenter transport over commodity network fabric, in: CONEXT, 2015, pp. 1–12.

[6] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, M. Sviridenko, Buffer overflow management in QoS switches, SIAM J. Comput. 33 (3) (2004) 563–583.

[7] I. Keslassy, K. Kogan, G. Scalosub, M. Segal, Providing performance guarantees in multipass network processors, IEEE/ACM Trans. Netw. 20 (6) (2012) 1895–1909.

[8] P. Chuprikov, S. I. Nikolenko, K. Kogan, Priority queueing with multiple packet characteristics, in: INFOCOM, 2015, pp. 1418–1426.

[9] D. D. Sleator, R. E. Tarjan, Amortized efficiency of list update and paging rules, Communications of the ACM 28 (2) (1985) 202–208.

[10] A. Borodin, R. El-Yaniv, Online Computation and Competitive Analysis, Cambridge University Press, 1998.

[11] C. T. C. A. for Internet Data Analysis, [Online] `http://www.caida.org/`.

[12] A. Davydow, P. Chuprikov, S. I. Nikolenko, K. Kogan, Throughput optimization with latency constraints, in: IEEE INFOCOM 2017 - IEEE Conference on Computer Communications, 2017, pp. 1–9.

[13] W. Aiello, Y. Mansour, S. Rajagopolan, A. Rosén, Competitive queue policies for differentiated services, in: INFOCOM, 2000, pp. 431–440.

[14] Y. Mansour, B. Patt-Shamir, O. Lapid, Optimal smoothing schedules for real-time streams, Distributed Computing 17 (1) (2004) 77–89.

[15] P. Chuprikov, S. I. Nikolenko, K. Kogan, On demand elastic capacity planning for service auto-scaling, in: INFOCOM, 2016, pp. 1–9.

[16] M. Goldwasser, A survey of buffer management policies for packet switches, SIGACT News 41 (1) (2010) 100–128.

[17] S. I. Nikolenko, K. Kogan, Single and multiple buffer processing, in: Encyclopedia of Algorithms, Springer, 2016, pp. 1988–1994.

[18] W. Aiello, A. Kesselman, Y. Mansour, Competitive buffer management for shared-memory switches, ACM Transactions on Algorithms 5 (1).

[19] Y. Azar, A. Litichevskey, Maximizing throughput in multi-queue switches, Algorithmica 45 (1) (2006) 69–90.

[20] N. Andelman, Y. Mansour, A. Zhu, Competitive queueing policies for QoS switches, in: SODA, 2003, pp. 761–770.

[21] A. Zhu, Analysis of queueing policies in QoS switches, J. Algorithms 53 (2) (2004) 137–168.

[22] A. Kesselman, Y. Mansour, R. van Stee, Improved competitive guarantees for QoS buffering, Algorithmica 43 (1-2) (2005) 63–80.

[23] J. Kawahara, K. Kobayashi, T. Maeda, Tight analysis of priority queuing policy for egress traffic, CoRR abs/1207.5959.

[24] P. T. Eugster, K. Kogan, S. I. Nikolenko, A. Sirotkin, Shared memory buffer management for heterogeneous packet processing, in: ICDCS, 2014, pp. 471–480.

[25] A. Kesselman, K. Kogan, M. Segal, Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing., Distributed Computing 23 (3) (2010) 163–175.

[26] A. Kesselman, K. Kogan, M. Segal, Best effort and priority queuing policies for buffered crossbar switches, Chicago J. Theor. Comput. Sci.

[27] A. Kesselman, K. Kogan, M. Segal, Improved competitive performance bounds for CIOQ switches, Algorithmica 63 (1-2) (2012) 411–424.

[28] K. Kogan, A. López-Ortiz, S. I. Nikolenko, A. V. Sirotkin, Online scheduling fifo policies with admission and push-out, Theory of Computing Systems 58 (2) (2016) 322–344.

[29] K. Kogan, A. López-Ortiz, S. I. Nikolenko, A. V. Sirotkin, The impact of processing order on performance: A taxonomy of semi-fifo policies, Journal of Computer and System Sciences 88 (2017) 220–235.

[30] K. Kogan, A. López-Ortiz, S. I. Nikolenko, A. Sirotkin, Multi-queued network processors for packets with heterogeneous processing requirements, in: COMSNETS, 2013, pp. 1–10.

[31] P. Eugster, K. Kogan, S. I. Nikolenko, A. V. Sirotkin, Heterogeneous packet processing in shared memory buffers, Journal of Parallel and Distributed Computing.

[32] E. L. Hahne, A. Kesselman, Y. Mansour, Competitive buffer management for shared-memory switches, in: SPAA, 2001, pp. 53–58.

[33] K. Kogan, A. López-Ortiz, S. I. Nikolenko, A. V. Sirotkin, A taxonomy of semi-FIFO policies, in: IPCCC, 2012, pp. 295–304.

[34] Z. Lotker, B. Patt-Shamir, Nearly optimal FIFO buffer management for two packet classes, Computer Networks 42 (4) (2003) 481–492.

[35] N. Andelman, Randomized queue management for diffserv, in: SPAA, 2005, pp. 1–10.

[36] M. Englert, M. Westermann, Lower and upper bounds on FIFO buffer management in QoS switches, Algorithmica 53 (4) (2009) 523–548.

[37] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S. I. Nikolenko, P. T. Eugster, BASEL (buffer management specification language), in: ANCS, 2016, pp. 69–74.

[38] K. Pruhs, Competitive online scheduling for server systems, SIGMETRICS Performance Evaluation Review 34 (4) (2007) 52–58.

[39] B. Malakooti, Operations and Production Systems with Multiple Objectives, John Wiley & Sons, 2013.

[40] J. A. Garay, Efficient on-line call control algorithms, J. Algorithms 23 (1) (1997) 180–194. `doi:10.1006/jagm.1996.0821`.
URL `http://dx.doi.org/10.1006/jagm.1996.0821`

[41] G. Koren, D. Shasha, Dover; an optimal on-line scheduling algorithm for overloaded real-time systems, in: [1992] Proceedings Real-Time Systems Symposium, 1992, pp. 290–299. `doi:10.1109/REAL.1992.242650`.

[42] C. Wilson, H. Ballani, T. Karagiannis, A. Rowtron, Better never than late: Meeting deadlines in datacenter networks, in: Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM '11, ACM, New York, NY, USA, 2011, pp. 50–61. `doi:10.1145/2018436.2018443`.
URL `http://doi.acm.org/10.1145/2018436.2018443`

[43] C.-Y. Hong, M. Caesar, P. B. Godfrey, Finishing flows quickly with preemptive scheduling, in: Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '12, ACM, New York, NY, USA, 2012, pp. 127–138. `doi:10.1145/2342356.2342389`.
URL `http://doi.acm.org/10.1145/2342356.2342389`

[44] V. Arun, H. Balakrishnan, Copa: Practical delay-based congestion control for the internet, in: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), USENIX Association, Renton, WA, 2018, pp. 329–342.
URL `https://www.usenix.org/conference/nsdi18/presentation/arun`

[45] B. Montazeri, Y. Li, M. Alizadeh, J. Ousterhout, Homa: A receiver-driven low-latency transport protocol using network priorities, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, ACM, New York, NY, USA, 2018, pp. 221–235. `doi:10.1145/3230543.3230564`.
URL `http://doi.acm.org/10.1145/3230543.3230564`

[46] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S. Nikolenko, A. Sirotkin, P. Eugster, A programmable buffer management platform, in: 2017 IEEE 25th International Conference on Network Protocols (ICNP), 2017, pp. 1–10.

[47] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, M. Ammar, K. Harras, A. Vahdat, Eiffel: Efficient and flexible software packet scheduling, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), USENIX Association, Boston, MA, 2019, pp. 17–32.
URL `https://www.usenix.org/conference/nsdi19/presentation/saeed`

[48] Ieee standard for a precision clock synchronization protocol for networked measurement and control systems, IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002) (2008) 1–300.