Throughput Optimization with Latency Constraints

Alex Davydow[*†], Pavel Chuprikov[†§], Sergey I. Nikolenko[†‡], Kirill Kogan[§]

* Saint Petersburg Academic University,
†Steklov Institute of Mathematics at St. Petersburg,
‡National Research University Higher School of Economics, St. Petersburg
§IMDEA Networks Institute, Madrid

Abstract—**Modern datacenters are increasingly required to deal with latency-sensitive applications. A major question here is how to represent latency in desired objectives. Incorporation of multiple traffic characteristics (e.g., packet values and required processing requirements) significantly increases the complexity of buffer management policies. In this work, we consider weighted throughput optimization (total transmitted value) in the setting where every incoming packet is branded with intrinsic value, required processing, and slack (an offset from the arrival time when a packet should be transmitted), and the buffer is unbounded but effectively bounded by slacks. The main result is a 3-competitive algorithm as the slack-to-work ratio increases. Our results supported by a comprehensive evaluation study on CAIDA network traces.**

I. 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 limitations [2], [8], [29].

Latency is certainly an important factor, but there are two primary ways to represent latency requirements in 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* [3], [15]. 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 [21] and called a *bounded-delay model*. In this model, the *slack* defines a latency constraint, i.e., the offset from 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.

However, the model of [21] 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 as deep packet inspection (DPI) and sophisticated virtual private network (VPN) services on the others; only recently this characteristic has become an object of study in admission control and scheduling decisions [9], [17]. 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 this work, we consider a single-queue switch where a buffer is shared among all types of traffic. We do not assume any specific traffic distribution but rather analyze our buffer management policies against adversarial traffic using competitive analysis [7], [34], which provides a uniform throughput guarantee for online algorithms under all possible arrival patterns. 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. In practice, the choices of processing

order, admission control logic are likely to be made at design time. From this point of view, our study of worst-case behaviour aims to provide a robust estimate on the settings that can handle all possible arrival patterns.

The paper is organized as follows. In Section II, we survey previous work in related buffer processing algorithms. Section III introduces a model that is used in this work. In Section IV, 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 V discusses a feasibility of a constant competitiveness and introduces a 3-competitive algorithm that is a main result of this paper. In Section VI we evaluate proposed algorithms on CAIDA traces [14]. Section VII concludes the paper.

II. RELATED WORK

There are two works [9], [21] that are closely related to this paper and consider weighted throughput (total transmitted value) as an optimized objective. In the model from [21], 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 [9] 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.

There has been a long line of work devoted to competitive analysis in models with only a single varying characteristic such as packet value [6], [18]–[20], [35] or required processing [12], [13], [17], [24]–[27] and with different buffer architectures. In particular, admission control policies for the case of a single-queue bounded buffer, where packets with uniform processing and varying intrinsic value arrive have been thoroughly studied. In the case of two values (1 and V) and First-In-First-Out (FIFO) processing order, the works [6], [35] present a deterministic non-push-out policy with a constant competitive ratio $(2 - \frac{1}{V})$. For the more general case, when intrinsic packet values vary between 1 and V, the works [6], [35] show that the competitive ratio cannot be better than $\Theta(\log V)$. In [5], the upper bound was improved to $2 + \ln V + O(\ln^2 V/B)$, where B is a buffer size. In the push-out case (when already admitted packets can be dropped) with two packet values, the greedy policy was shown in [22] to be at least 1.282 and at most 2-competitive. Later, the upper bound on the greedy policy was improved to 1.894 [23]. Policies with memory have been considered in [4], [11], [30].

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 [1], [22], [32], the competitive analysis for buffer management has received tremendous attention over the past decade. The bounded-delay model [21] is shown to be effective for service scale-outs [10]. The surveys by Goldwasser [16], Nikolenko and Kogan [33] provide a comprehensive overview of this field. The language to express buffer managment policies for user-defined policies is proposed in [28].

III. MODEL DESCRIPTION

We use a model similar to the one introduced in [21]. Consider a single queue that handles the arrival of a sequence of unit-sized packets. While previous works [9], [21] have dealt with the case of two characteristics (value and slack in [21], value and required processing in [9]), 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* $s \in \{1, \ldots, S\}$ 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 [21], 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 [21].

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.

IV. IMPACT OF PROCESSING ORDERS

It is well known that different processing orders have a significant impact on the performance of buffer management policies [9], [17]. 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 [21], a queue has an unbounded size, so during admission we can accept all packets, later automatically discarding packets with expired slack, so we only need to define the processing order. 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[1]. In what follows, we introduce several processing orders to understand their impact on weighted throughput (total transmitted value).

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.

---

[1]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.
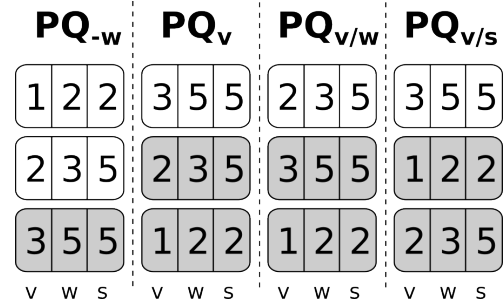

Fig. 1. A sample time slot for PQ_{-w}, PQ_v, $PQ_{v/w}$, and $\mathrm{PQ}_{v/s}$. The head of line packet is at the top. Shaded packets are those that will not be transmitted due to lack of time.

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 [31].

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.

Finally, we proceed to establish a *general lower bound*; 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. 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 work 1 and another with value V and work $W = V^2$. 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 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". In our main result, we will see that if we require the slack s to be separated from w by at least any constant multiplicative factor, 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 [9], those were proven for a bounded buffer and hence are inapplicable in our case. Theorem 1 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 2. *If w, s, and v are admissible values of work, slack, and value respectively, 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*

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

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 $s \geq w$, 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 $\frac{s}{w}$, so the total processed value can be bounded from above by $v(\frac{w}{w-2})^n \frac{s}{w}$. 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 for any fixed $\frac{w}{s}$ it tends to infinity if v and w tend to infinity. ∎

Theorem 2 shows a common problem of priority queue algorithms: they abandon packets they have already invested in, and thus sometimes a large part of the processing time is wasted. One possible approach to solve this problem is to introduce the so-called β-*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 $f(p') \geq \beta f(p)$. For $\beta > 1$, this introduces a commitment to already processed packets, helping to solve the above problem, and we will see them in experimental evaluation. However, in the worst case β-preemption also do not really help; we illustrate it below for $\mathrm{PQ}_{v/w}$, and similar examples can be constructed for other priority queues.

Theorem 3. *The $\mathrm{PQ}_{v/w}$ algorithm with β-preemption has a non-constant lower bound for arbitrarily large slacks.*

4

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 arrives 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 $\frac{\beta+1}{w}v$, 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 $cv + \frac{(k-c+1)v(\beta+1)}{w}$ while another algorithm could process only large packets for a total value of kv which can be greater than $cv + \frac{(k-c+1)v(\beta+1)}{w}$ 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). ∎

While algorithms with β-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 3). The second problem is that while preemption does lend some stability and robustness to the algorithm, once a preemptive algorithm has abandoned processing of some packet it is not going to prefer it to unprocessed packets, wasting invested time in vain. In the next section we present an algorithm that is free of these problems and identify conditions when it has a constant competitive ratio.

V. CONSTANT COMPETITIVENESS WITHOUT JUST-IN-TIME PACKETS

We have seen in Theorem 1 that any deterministic online algorithm is non-competitive in the general case; however, as we have noted, the counterexample relies upon the availability of "just-in-time" packets with $s = w$. Surprisingly, as soon as we require any, however small separation between w and s, we can immediately 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.

Definition V.1. *Algorithm* SPQ *(stack with priority queue) operates as follows: apart from the queue, it*
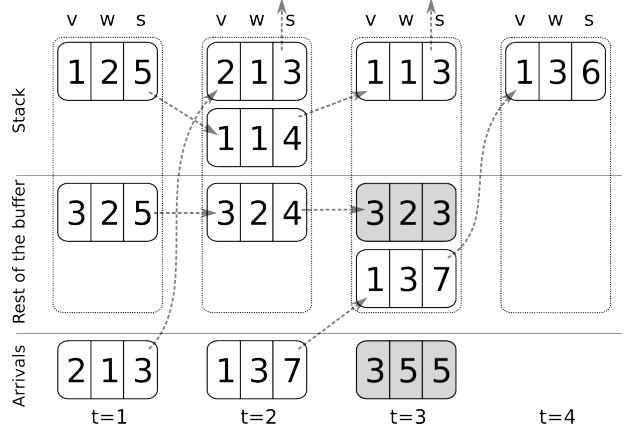

Fig. 2. Sample operation of the SPQ algorithm.

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[2], denoted* HOL *with parameters* $[\![w_h \mid v_h \mid s_h]\!]$; *we also denote by* $\mathrm{HOL}^t = [\![w_h^t \mid v_h^t \mid s_h^t]\!]$ *the* HOL *packet at time 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 $s < (1 + \beta)w$ (the "freelancer 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;*

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

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 freelancer 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 freelancer

---

[2]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.

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.

Theorem 4. *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 is summarized in the following lemma.

Lemma 5. *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, for a total of $\leq \frac{1}{\alpha} + \frac{1}{\alpha^2} + \frac{1}{\alpha^3} + \ldots = \frac{1}{\alpha - 1}$ of the total value.

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 $1 - \frac{1}{(\alpha - 1)\beta}$. ∎

We now proceed to proving the theorem. From this moment on, we will consider bites instead of full packets, and by Lemma 5 we can simply multiply the resulting upper bound by $\delta := \frac{1}{1 - \frac{1}{(\alpha - 1)\beta}} = \frac{\beta(\alpha - 1)}{\beta\alpha - \beta - 1}$.

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:
   a) at the time when OPT was processing it p has not been accepted on stack by (R2);
   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 futher 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 sum of ratios from cases 1–3.

In case 1, we match the bites one-to-one, and they are covered by Lemma 5, 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 5 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_{arr}^p the time when packet p arrives, by t_{beg}^p the time when OPT begins processing p, and by t_{end}^p the time when OPT finishes processing p. We sort the packets in the order of t_{end}.

Lemma 6. *Consider a stream of packets $(p_1, p_2, \ldots)$ sorted by t_{end}^p. We construct the mapping as follows: for each $p = [\![w \mid v \mid s]\!]$, we map it to an arbitrary subset of $\frac{1}{\alpha}w$ unmapped* SPQ *bites from the interval $[t_{\text{arr}}^p, t_{\text{beg}}^p]$. Then this mapping 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 $(\gamma + 1 + \beta)w \leq s$. 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 $[t_{\text{arr}}^p, t_{\text{end}}^p]$. First, there are at least $s - (1 + \beta)w \geq \gamma w$ 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 $[t^p_{\mathrm{arr}}, t^p_{\mathrm{beg}}]$ can only be mapped to packets fully processed by OPT over the interval $[t^p_{\mathrm{arr}}, t^p_{\mathrm{end}}]$: if a packet p' has $t^{p'}_{\mathrm{end}} > t^p_{\mathrm{end}}$, it comes later in the ordering, and if it has $t^{p'}_{\mathrm{beg}} < t^p_{\mathrm{arr}}$ 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 $t^p_{\mathrm{end}} - t^p_{\mathrm{arr}} = s$, and OPT needs w timeslots to actually process p; hence at most $\frac{\gamma}{c}(s-w)$ SPQ bites from $[t^p_{\mathrm{arr}}, t^p_{\mathrm{end}}]$ 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 $\gamma w - \frac{\gamma}{c}(s - w) \geq \gamma w \left(1 - \frac{1}{c}(c - 1)\right) \geq \frac{\gamma}{c} w$ bites free to be mapped to p, which proves that the mapping is always feasible. $\blacksquare$

By Lemma 6, 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

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

$\blacksquare$

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 $c \to \infty$.

VI. Evaluation

To evaluate our policies, we have used real traffic traces from CAIDA (Center for Applied Internet Data Analysis) [14]. 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 a size of time slot determines traffic burstiness even for fixed packet traces.

The traces provide traffic distributions for the incoming packets: we specify a size of time slot and take all packets in this window to arrive at the same time slot.
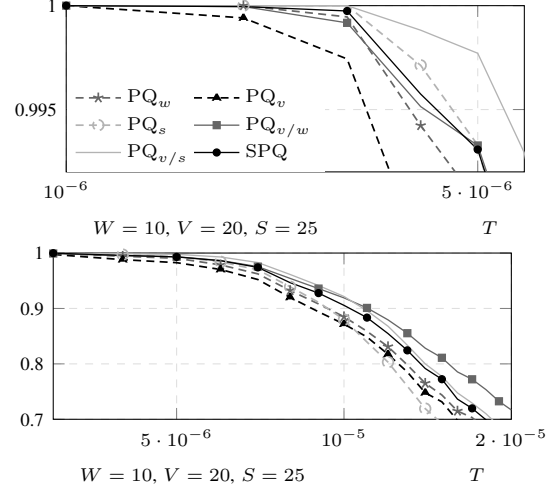

Fig. 3. Share of transmitted value as a function of time slot size T for $W = 10$, $V = 20$, $S = 25$.
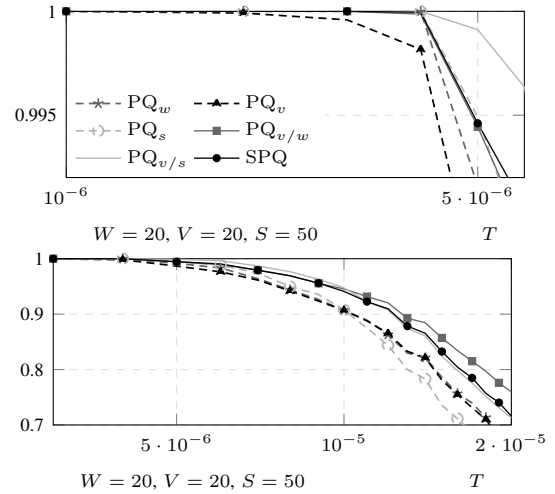

Fig. 4. Share of transmitted value as a function of time slot size T for $W = 20$, $V = 20$, $S = 50$.

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.[3]

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

[3]We have made the source code for our simulations available on http://github.com/submissioninfocom/throughput-latency; naturally, the repository does not contain CAIDA traces.
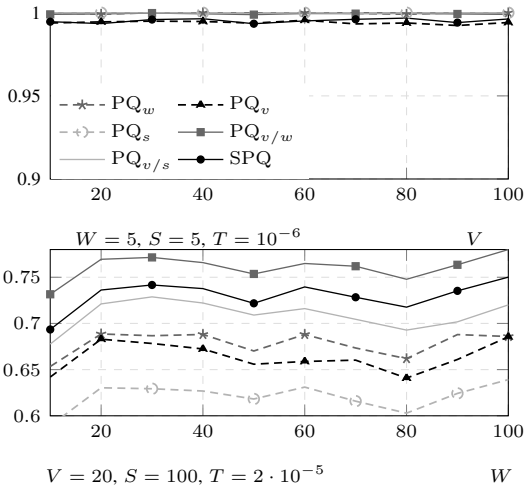
Fig. 5. Share of transmitted value as a function of maximal value V and maximal work W.
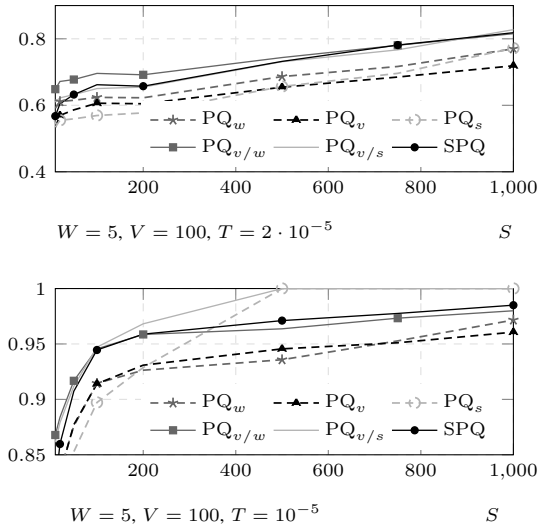

Fig. 6. Share of transmitted value as a function of maximal slack S.

the size of time slot (in seconds) for the CAIDA traces. The size of time slot has a direct linear relationship with the unit intensity of the incoming stream of packets λ: a larger size of 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 50,000 time slots in order to obtain a clear and robust evaluation.

Figures 3 and 4 show the main results of our simulation study: the dependence of the share of transmitted value on the time slot size, that is, on the intensity of the incoming stream of packets. In our experiments, the most interesting effects happen when the packet intensity $\frac{1}{2}\lambda/W$ is close to 1, i.e., packets arrive with relatively little congestion. Our results indicate that overall, the best policies are SPQ, $\mathrm{PQ}_{v/w}$, and $\mathrm{PQ}_{v/s}$, which have more processed value compared to other policies. When $\frac{1}{2}\lambda/W$ is close to 1, and thus the share of processed packets is 1 or close to 1, $\mathrm{PQ}_{v/s}$ performs best, and SPQ is in second place. However, as the unit density grows $\mathrm{PQ}_{v/w}$ takes the lead, and $\mathrm{PQ}_{v/s}$ drops to the third place; throughout the ranges of parameters in our experiments, SPQ held a steady second place. In order to show this effect, Figs. 3 and 4 show the same plots on two different scales, with an enlarged version on the left.

Figure 5 shows sample graphs from the second set of simulations, where we investigated how the share of transmitted value depends on V and W. As expected, throughout the entire range of values in our simulation we have noticed no significant dependence on either V or W.

Figure 6 shows how the share of transmitted value depends on the maximal slack S. Here we show two different settings: in the first (top) plot, we see a relatively congested situation with all algorithms transmitting around 0.6–0.8 of the total possible value. In this setting, SPQ and $\mathrm{PQ}_{v/w}$ work best, and SPQ overcomes $\mathrm{PQ}_{v/w}$ as the slack grows (and thus congestion decreases). On the bottom, we see a relatively uncongested situation (with twice smaller time slot size), and here the leaders are completely different: PQ_{s} and $\mathrm{PQ}_{v/s}$ are the first to achieve perfect transmission. Note, however, that in both settings SPQ performs well.

VII. 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 [14] 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 function to be optimized, as in the average flow completion time model.

REFERENCES

[1] William Aiello, Yishay Mansour, S. Rajagopalan, and Adi Rosén. Competitive queue policies for differentiated services. In *INFO-COM*, pages 431–440, 2000.

[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX*, pages 281–296, 2010.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: minimal near-optimal datacenter transport. In *SIGCOMM*, pages 435–446, 2013.

[4] Nir Andelman. Randomized queue management for diffserv. In *SPAA*, pages 1–10, 2005.

[5] Nir Andelman and Yishay Mansour. Competitive management of non-preemptive queues with multiple values. In *DISC*, pages 166–180, 2003.

[6] Nir Andelman, Yishay Mansour, and An Zhu. Competitive queueing policies for QoS switches. In *SODA*, pages 761–770, 2003.

[7] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[9] Pavel Chuprikov, Sergey I. Nikolenko, and Kirill Kogan. Priority queueing with multiple packet characteristics. In *INFOCOM*, pages 1418–1426, 2015.

[10] Pavel Chuprikov, Sergey I. Nikolenko, and Kirill Kogan. On demand elastic capacity planning for service auto-scaling. In *INFOCOM*, pages 1–9, 2016.

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

[12] Patrick Th. Eugster, Alexander Kesselman, Kirill Kogan, Sergey I. Nikolenko, and Alexander Sirotkin. Essential traffic parameters for shared memory switch performance. In *SIROCCO*, pages 61–75, 2015.

[13] Patrick Th. Eugster, Kirill Kogan, Sergey I. Nikolenko, and Alexander Sirotkin. Shared memory buffer management for heterogeneous packet processing. In *ICDCS*, pages 471–480, 2014.

[14] CAIDA The Cooperative Association for Internet Data Analysis. [Online] http://www.caida.org/.

[15] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CONEXT*, pages 1–12, 2015.

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

[17] Isaac Keslassy, Kirill Kogan, Gabriel Scalosub, and Michael Segal. Providing performance guarantees in multipass network processors. *IEEE/ACM Trans. Netw.*, 20(6):1895–1909, 2012.

[18] Alexander Kesselman, Kirill Kogan, and Michael Segal. Improved competitive performance bounds for CIOQ switches. In *ESA*, pages 577–588, 2008.

[19] Alexander Kesselman, Kirill Kogan, and Michael Segal. Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing. *Distributed Computing*, 23(3):163–175, 2010.

[20] Alexander Kesselman, Kirill Kogan, and Michael Segal. Best effort and priority queuing policies for buffered crossbar switches. *Chicago J. Theor. Comput. Sci.*, 2012, 2012.

[21] Alexander Kesselman, Zvi Lotker, Yishay Mansour, Boaz Patt-Shamir, Baruch Schieber, and Maxim Sviridenko. Buffer overflow management in QoS switches. In *STOC*, pages 520–529, 2001.

[22] Alexander Kesselman, Zvi Lotker, Yishay Mansour, Boaz Patt-Shamir, Baruch Schieber, and Maxim Sviridenko. Buffer overflow management in QoS switches. *SIAM Journal on Computing*, 33(3):563–583, 2004.

[23] Alexander Kesselman and Yishay Mansour. Loss-bounded analysis for differentiated services. *J. Algorithms*, 46(1):79–95, 2003.

[24] Kirill Kogan, Alejandro López-Ortiz, Sergey I. Nikolenko, Gabriel Scalosub, and Michael Segal. Large profits or fast gains: A dilemma in maximizing throughput with applications to network processors. *J. Network and Computer Applications*, 74:31–43, 2016.

[25] Kirill Kogan, Alejandro López-Ortiz, Sergey I. Nikolenko, and Alexander Sirotkin. Multi-queued network processors for packets with heterogeneous processing requirements. In *COMSNETS*, pages 1–10, 2013.

[26] Kirill Kogan, Alejandro López-Ortiz, Sergey I. Nikolenko, and Alexander V. Sirotkin. A taxonomy of semi-FIFO policies. In *IPCCC*, pages 295–304, 2012.

[27] Kirill Kogan, Alejandro López-Ortiz, Sergey I. Nikolenko, and Alexander V. Sirotkin. Online scheduling FIFO policies with admission and push-out. *Theory Comput. Syst.*, 58(2):322–344, 2016.

[28] Kirill Kogan, Danushka Menikkumbura, Gustavo Petri, Youngtae Noh, Sergey I. Nikolenko, and Patrick Th. Eugster. BASEL (buffer management specification language). In *ANCS*, pages 69–74, 2016.

[29] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[30] Zvi Lotker and Boaz Patt-Shamir. Nearly optimal FIFO buffer management for two packet classes. *Computer Networks*, 42(4):481–492, 2003.

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

[32] Yishay Mansour, Boaz Patt-Shamir, and Ofer Lapid. Optimal smoothing schedules for real-time streams. *Distributed Computing*, 17(1):77–89, 2004.

[33] Sergey I. Nikolenko and Kirill Kogan. Single and multiple buffer processing. In *Encyclopedia of Algorithms*, pages 1988–1994. Springer, 2016.

[34] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[35] An Zhu. Analysis of queueing policies in QoS switches. *J. Algorithms*, 53(2):137–168, 2004.

9