The Impact of Processing Order on Performance: a Taxonomy of Semi-FIFO Policies

Kirill Kogan[a,*], Alejandro López-Ortiz[b], Sergey I. Nikolenko[c,d], Alexander V. Sirotkin[c,e]

[a] *IMDEA Networks Institute*
[b] *Department of Computer Science, University of Waterloo*
[c] *National Research University Higher School of Economics, Russia*
[d] *Steklov Mathematical Institute, St. Petersburg*
[e] *St. Petersburg Institute for Informatics and Automation of the RAS*

Abstract

Modern network processors increasingly deal with packets that require heterogeneous processing. We consider the problem of managing a bounded size input queue buffer where each packet requires several rounds of processing before it can be transmitted out. This to maximize the total number of successfully transmitted packets. Usually the transmission order of the packets is induced by the processing order. However, processing order can have a significant impact on the performance of buffer management policies even if the order of transmission is fixed. For this reason we decouple processing order from transmission order and restrict our transmission order to First-In-First-Out (FIFO) but allow for different orders of packet processing, introducing the class of such policies as Semi-FIFO. In this work, we build a taxonomy of Semi-FIFO policies and provide worst case guarantees for different processing orders. We consider various special cases and properties of Semi-FIFO policies, e.g., greedy, work-conserving, lazy, and push-out policies, and show how these properties affect performance. We generalize our results to additional constraints on the push-out mechanism designed to deal with copying cost. Further, we conduct a comprehensive simulation study that validates our results[1].

*Corresponding author

Email addresses: kirill.kogan@imdea.org (Kirill Kogan), alopez-o@uwaterloo.ca (Alejandro López-Ortiz), sergey@logic.pdmi.ras.ru (Sergey I. Nikolenko), avsirotkin@hse.ru (Alexander V. Sirotkin)

[1] A preliminary version of this paper appeared in [1]. Here is a list of the main changes made as compared to the conference version.

1. We have completely reworked the simulations section (Section 6) and have redone the experiments.
2. We have proven a new lower bound on β-preemptive policies (Theorem 14).
3. We have extended and augmented the introduction and discussion sections.

1. Introduction

Network processors are widely used to perform packet processing tasks in modern high-speed routers. They are often implemented with multiple processing cores. Such architectures are very efficient for simple traffic profiles.

The modern network edge is required to perform tasks with heterogeneous complexity, including features such as advanced VPNs services, deep packet inspection, firewall, intrusion detection (to list just a few). Each of these features may require a different amount of processing at the network processors and may introduce new challenges for traditional architectures, posing implementation, fairness, and performance issues. All of these features directly affect processing delay. As a result, the processing order of packets and the way how these packets are processed can have a significant impact on the queueing delay and throughput; increasing the required processing per packet for some of the flows can cause increased congestion even for traffic with relatively modest burstiness characteristics. As [2] showed, processing requirements on a network processor can be approximately predicted for a given configuration.

The main purpose of this paper is to improve the understanding of the effect of composing processing order with various admission control policy properties on throughput at the network processor. Though goodput precisely represents actual transferred data, intermediate network elements are mostly unaware about end-to-end decisions of transports. In such settings throughput is usually used to estimate performance of network elements. Here, we consider cases, where the buffer is bounded and arrivals require heterogeneous processing. Our goal is to conduct for a systematic study of buffer management policies for packets with heterogeneous processing. Although we present a comprehensive simulation study to validate our results, the main emphasis of this work is on worst-case performance guarantees for various classes of buffer management policies.

There is one important question we have to address: why consider worst-case performance guarantees instead of estimating statistical guarantees under stochastic assumptions on properties of arrivals? The properties of heterogeneous processing depend on applied features that can vary significantly between different locations of the processor in the network. Different processing orders can perform differently under different distributions of required processing. Although some parameters of admission control policies can be configurable, it is unlikely that a buffer implementation can have a configurable processing order. So instead of approximating properties of arrivals we compare performance of various buffer management policies without any assumptions on specific traffic and required processing distributions. An additional advantage of such an approach is that it lets us estimate the required speedup factor (in our case the number of cores) to guarantee close to optimal performance for any type of arrivals.

1.1. Model Description

In what follows, we adopt the terminology used to describe buffer management problems [3, 4] and focus most of our attention on a family of *Semi-FIFO* policies that may process packets in different orders but transmit them in FIFO (First-In-First-Out) order. In our system, arriving traffic consists of unit-sized *packets*, and each packet has a *processing requirement* (in processor cycles). We consider a buffer with

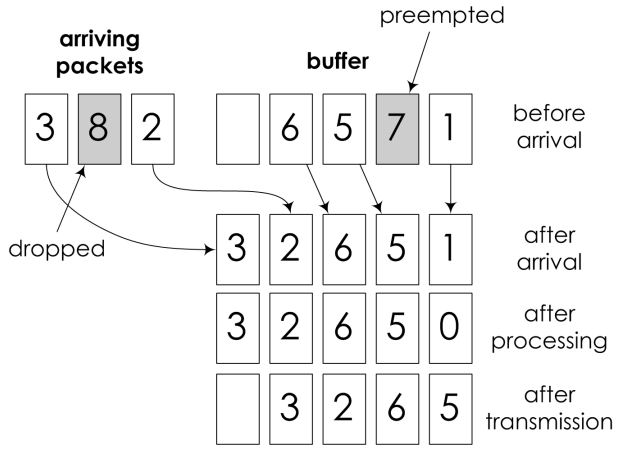

Figure 1: Zoom in on one time slot.

bounded capacity B that handles the arrival of a sequence of unit-sized packets. Each arriving packet p is branded with the number of required processing cycles $r(p) \in \{1, \ldots, k\}$. This number is known for every arriving packet; for a justification of why such information can be assumed to be available see [2]. We will denote a packet with required processing r by $\boxed{r}$, and a sequence of n packets with the same required processing r by $n \times \boxed{r}$. Although the value of k will play a fundamental role in our analysis, our algorithms will not need to know k in advance.

The queue performs three main tasks, namely:

(i) *buffer management*, i.e., admission control of newly arrived packets and push-out of currently stored packets (if push-out capability is supported);

(ii) *processing*, i.e., deciding which of the currently stored packets will be processed;

(iii) *transmission*, i.e., deciding if already processed packets should be transmitted.

A packet is *fully processed* if the processing unit has scheduled the packet for processing for at least its required number of cycles. Our framework assumes a multi-core environment with C cores, and at most C packets may be chosen for processing at any given time. However, since we consider specific processing orders, a new mechanism to reorder simultaneously processed packets is required. For simplicity, in the remainder of this paper we assume that a policy selects a single packet for processing and at most one packet for transmission at any given time (i.e., $C = 1$). This simple setting suffices to show our main results and the difficulties of the considered model; we consider multiple cores in simulations.

We assume discrete slotted time, where each time slot consists of three phases (see Fig. 1 for an example):

(i) *arrival*: new packets arrive, and the admission control decides if it should be dropped or, possibly, push-out an already admitted packet;

(ii) *processing*: a single 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 (i.e., while it still has a positive number of required processing cycles), it is lost. Note that 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 one unit to the objective function only upon being successfully transmitted. The goal is to devise buffer management algorithms that maximize the overall throughput, i.e., the total number of packets transmitted from the queue. As a natural generalization, in Section 5 we consider a different objective function that assigns copying costs for a packet to enter the buffer.

For an algorithm ALG and a time slot t, we denote the set of packets stored in A's buffer at time t by $\mathrm{IB}_t^{\mathrm{ALG}}$. The number of *processing cycles* of a packet is key to our algorithms. Formally, for every time slot t and every packet p currently stored in the queue, the number of *residual processing cycles* (residual work), denoted $r_t(p)$, is the number of processing cycles it requires before it can be successfully transmitted.

We use competitive analysis [5, 6] to evaluate performance guarantees provided by our online algorithms. An algorithm ALG is said to be α-*competitive* (for some $\alpha \geq 1$) if for any arrival sequence σ the number of packets successfully transmitted by ALG is at least $1/\alpha$ times the number of packets transmitted in an optimal solution (denoted OPT) obtained by an offline clairvoyant algorithm. The value of competitive ratio can also be viewed as the value of speedup necessary to achieve equivalent-to-optimal performance in the network processor. Note that a *lower bound* on the competitive ratio can be proven with a specific hard example while an *upper bound* represents a general statement that should hold over all possible inputs.

In this paper, we sometimes compare the proposed policies directly in the worst case. An algorithm A_1 *outperforms* an algorithm A_2 if for any input σ A_1 transmits at least the number of packets that A_2 transmits. Otherwise, A_1 and A_2 are *incomparable* in the worst case. It turns out that, perhaps unsurprisingly, most meaningful policies are incomparable with each other in the worst case. This is an additional factor that demonstrates the importance of considering worst-case guarantees for various processing orders: for some traffic and required processing distributions processing orders with relatively bad worst-case performance guarantees can perform better than good (the sense of worst-case performance) processing orders. Observe that in practice the choices of processing order, implementation of push-out mechanisms etc. are likely to be made at design time. In this sense our study of worst-case behaviour aims to provide a robust estimate on the settings that can handle all possible loads.

1.2. Related Work

Keslassy et al. [7] were the first to consider buffer management and scheduling in the context of network processors with heterogeneous processing requirements for arriving traffic. They study both PQ (Priority Queue) and FIFO schedulers with recycles, in both push-out and non-push-out buffer management cases, where a packet is recycled after processing according to the priority policy. For the case of FIFO with recycles, only preliminary results were obtained. A major result of [7] is the introduction of the copying cost model where each admitted packet incurs some intrinsic copying

cost $\alpha > 0$ in the context of PQ processing order. Kogan et al. [4] considered the FIFO case for packets with heterogeneous processing and proved several upper and lower bounds for the proposed algorithms.

Our present work can be viewed as part of a larger research effort concentrated on studying competitive algorithms with buffer management for bounded buffers (see, e.g., recent surveys [8, 9] that provides an excellent overview of this field). This line of research, initiated in [10, 11], has received tremendous attention over the last decade.

Various models have been proposed and studied, including, among others, QoS-oriented models where packets have values [10, 11, 12, 13] and models where packets have dependencies [14]. A related field focuses on various switch architectures and aims to design competitive algorithms for such multi-queue scenarios; see, e.g., [15, 16, 17, 18, 19, 20, 21]. Some other works also provide experimental studies of these algorithms and further validate their performance [22]. The work of [23] considers applications of an online learning approach to buffer management problems. More recently different works [24, 25, 26] study throughtput optimization with multiple packet charachteristics for a single queue buffering architecture.

There is a long history of OS scheduling for multithreaded processors which is relevant to our research. For instance, the PQ processing order has been studied extensively in such systems, and it is well known to be optimal with respect to mean response [27]. Additional objectives, models, and algorithms have been studied in this context [28, 29, 30]. For a comprehensive overview of competitive online scheduling for server systems, see a survey by Pruhs [31]. One should note that OS scheduling is mostly concerned with average response time, but we focus on estimating the throughput. Furthermore, OS scheduling does not allow jobs to be dropped, which is an inherent aspect of our proposed model since we have a limited-size buffer. [32] introduce a language to specify buffer managment policies for user-defined buffering architectures.

1.3. Our Contributions

In a number of ways, this paper generalizes previous results of Keslassy et al. [7]. However, the primary goal of this work is different; we aim to provide a systematic study of buffer management policies for packets with heterogeneous processing. We consider two major factors that affect performance of buffer management policies. The first factor is the processing order of packets that may differ from the transmission order. To demonstrate the impact of processing order on performance, we define a class of *Semi-FIFO* polices that can process packets in any order, but their transmission order is restricted to FIFO. For this purpose, we consider the class of *lazy* algorithms that delay packet transmission if the buffer still contains packets that require more than one processing cycle. Once all packets have been nearly processed, such lazy algorithms flush out the buffer in FIFO order. Moreover, we consider *greedy* policies (that accept everything if there is free buffer space) and *push-out* policies (that are allowed to drop already accepted packets) and prove a general upper bound which is independent of the processing order. We also demonstrate that for some processing orders this upper bound is tight. In addition, we show lower and upper bounds for other processing orders which are significantly stronger than previous results. These results are presented in Section 3.

Next, we return to policies where transmission order corresponds to processing order and show how performance guarantees for Semi-FIFO policies can be applied to such policies. In Section 4, we compare lazy and non-lazy policies with the same processing and transmission order. We show that for some orders, these algorithms are not comparable in the worst case; this is a counterintuitive result since non-lazy algorithms usually tend to free their buffer as fast as possible, faster than lazy ones. Understanding that in some systems delaying packet transmission may be unacceptable, we demonstrate how to construct non-lazy algorithms that have the same performance guarantees as lazy ones.

In Section 5, we consider different constraints on the push-out mechanism of admission control. In addition to the well-known non-push-out type, we consider two different types of such constraints: *additive* (that forbid push-out of already partially processed packets) and *multiplicative* (when a new packet p pushes out an accepted packet q only if the required processing for q is at least β times more than the required processing of p for some $\beta > 1$). In our analysis of the multiplicative constraint, we analyze the impact of *copying cost* on the performance of Semi-FIFO polices: copying cost $\alpha > 0$ means that for each packet copied to the buffer the algorithm pays a penalty of α. This results in a different objective function: instead of the total number of packets, maximize the transmitted value, gaining 1 for every successfully transmitted packet and losing α for every admitted one. We show that copying cost (originally introduced in [7] for the PQ order) is a powerful mechanism to control the number of pushed out packets regardless of processing order. We provide a general upper bound for greedy push-out Semi-FIFO policies. Over subsequent sections, we show the advantages and universality of the "lazy" infrastructure for the analysis of buffer management algorithms as compared with ad-hoc methods mostly used before. A major result of our study is a taxonomy of Semi-FIFO policies. In Section 6, we show the results of a comprehensive simulation study that validates our theoretical contributions.

2. Properties of Semi-FIFO Policies

In this section, we define the set of properties incorporated into our classification of Semi-FIFO policies. The first property is the capability to *push out* already accepted packets. In [7], it was shown that policies with push-out significantly outperform non-push-out FIFO with recycles. The second property is *greediness*. Greedy algorithms are usually amenable to efficient implementation and transmit everything if there is no congestion, though we will demonstrate that in some cases greediness can cause performance to drop. The third property is *laziness*; a policy is *lazy* if it does not send a packet while there is at least one packet in the buffer that requires more than one processing cycle. Laziness is possibly the most interesting property since it allows for a well-defined accounting infrastructure. Surprisingly, in some cases lazy algorithms outperform algorithms with the same processing order that tend to send out packets as soon as their processing is completed. Fourth, a policy is called *work-conserving* if it always transmits fully processed packets. A policy is usually defined by its processing order and a subset of the above properties. In this paper, we consider the following processing orders:

(1) *priority queueing* (PQ), where a packet with minimal residual work is processed first;

(2) *reversed priority queueing* (RevPQ), where a packet with maximal residual work is processed first;

(3) *FIFO*, a regular first-in-first-out processing order;

(4) *FIFO with recycles* (RFIFO), where non-fully processed packets are recycled to the back of the queue after each processing step.

3. Impact of Processing Order on Performance

In this section, we consider push-out Semi-FIFO algorithms. We will consider additional restrictions on the push out mechanism in Section 5. For now, if upon the arrival of a packet p the buffer of an online policy is full and p requires strictly less processing than at least one packet in the buffer, the first (according to the processing order) admitted packet with maximal required work is pushed out and p is accepted to the buffer according to processing order defined by the policy. If not explicitly specified otherwise, in what follows all algorithms are *greedy* (i.e. they accept any packet if there is available buffer space).

3.1. Lazy Push-Out Policies

In this section, we consider an interesting class of policies, namely *lazy* buffer management policies, and prove a general upper bound for all processing orders in lazy policies.

Definition 3.1. *A buffer processing policy* LA *is called* lazy *if it satisfies the following conditions:*

(1) LA *greedily accepts packets if its buffer is not full;*

(2) LA *pushes out the first packet with maximal number of processing cycles in case of congestion;*

(3) LA *does not process and transmit packets with a single processing cycle if its buffer contains at least one packet with more than one processing cycle left;*

(4) once all packets in LA*'s buffer (say* m *packets) have a single processing cycle remaining,* LA *transmits them over the next* m *time slots, even if additional packets arrive (and are accepted) during that time.*

The definition of LA allows for a well-defined accounting infrastructure. In particular, LA's definition lets us define an *iteration* over which we can count the number of packets transmitted by the optimal algorithm and compare it to the contents of LA's buffer. The first iteration begins with the first arrival. An iteration ends when all packets in the LA buffer have a single processing pass left. Each subsequent iteration starts after LA has transmitted all packets from the previous iteration. Since all packets at the

end of an iteration have a single processing cycle, these packets can be transmitted in any order (in our case FIFO). Note that Definition 3.1 covers a wide array of policies because it does not specify the processing order during an iteration. This actually does matter since processing order determines how much processing is "wasted" on pushed out packets.

A lazy push out policy is, to a certain extent, a theoretical artifact intended to simplify the analysis of various policies. In practice one is likely to implement non-lazy policies that emulate lazy policies in most other aspects and thus their practical performance is no worse and likely better than that guaranteed by the corresponding analysis of its lazy counterpart. However, we will see in Theorem 8 that these policies tend to be incomparable in the worst case.

Next we show one of the main results of this work: an upper bound for any lazy policy regardless of its processing order.

Theorem 1. LA *is at most* $\left(\frac{1}{B}\log_{B/(B-1)} k + 3\right)$*-competitive.*

Note that asymptotically, for $k, B \to \infty$, this is a logarithmic bound in k:

$$\frac{1}{B}\log_{B/(B-1)} k = \frac{\ln k}{B\ln\left(1+\frac{1}{B}+o(\frac{1}{B})\right)} = \ln k + o(\ln k).$$

For this purpose, we assume that the optimal algorithm OPT never pushes out packets and is work-conserving; without loss of generality, every optimal algorithm can be assumed to have these properties since the input sequence is available for it *a priori*. Further, we enhance OPT with two additional properties:

(1) at the start of each iteration, OPT flushes out all packets remaining in its buffer from the previous iteration (for free, with extra gain to its throughput);

(2) let t be the first time when LA's buffer is congested during an iteration; OPT flushes out all packets that currently reside in its buffer at time $t-1$ (again, for free, with extra gain to its throughput).

Clearly, the enhanced version of OPT is no worse than the optimal algorithm. In what follows, we will compare LA with this enhanced version of OPT for the purposes of an upper bound. Note that although all algorithms we consider in this work are semi-FIFO (i.e., with FIFO transmission order), our upper bounds are also valid for OPT which is not semi-FIFO; on the other hand, we do not show any better bounds for semi-FIFO OPT; this remains an interesting open problem.

To avoid ambiguity in reference time, t should be interpreted as the arrival time of a single packet. If more than one packet arrive at the same time slot, this notation is considered for each packet independently, in the sequence in which they arrive (although they might share the same actual time slot). In what follows, *buffer occupancy* is defined as the number of packets residing in the buffer at a given time.

Lemma 2. *Consider an iteration I that begins at time t' and ends at time t, i.e., spans the interval $[t', t]$. The following statements hold.*

1. *During I, at any time before* LA *begins transmission the buffer occupancy of* LA *is at least the buffer occupancy of* OPT.

2. *Between two consecutive iterations I and I',* OPT *transmits at most* $|\,\mathrm{IB}_t^{\mathrm{LA}}\,|$ *packets.*

3. *If during a time interval* $[t', t"] \subseteq [t', t]$ *there is no congestion, then during* $[t', t"]$ OPT *transmits at most* $|\,\mathrm{IB}_{t"}^{\mathrm{LA}}\,|$ *packets.*

Proof. 1. By definition of OPT and since LA always acts greedily unless its buffer is full.

2. By (1), at the end of an iteration the buffer occupancy of LA is at least the buffer occupancy of OPT; moreover, all packets in LA's buffer at the end of an iteration have a single processing cycle remaining.

3. Since during $[t', t"]$ there is no congestion and since LA is greedy, LA's buffer contains all packets that have arrived after t; obviously, OPT cannot transmit more packets than have arrived.

□

Let us now go over the iteration and count the packets that OPT can process on each step. We denote by t_{beg} the time when iteration I begins; by t_{con}, the time of the first congestion in LA's buffer; by t_{end}, the time when I ends. We start the iteration with an empty OPT buffer, so during $[t_{\mathrm{beg}}, t_{\mathrm{con}}]$ OPT obviously processes at most $B-1$ packets: since LA does not transmit any packets and becomes congested at t_{con}, there arrive at most B packets during this interval, and, moreover, OPT cannot process all B packets because then LA would also have time to process them, and the iteration would be uncongested.

We denote by M_t the maximal number of residual processing cycles among all packets in LA's buffer at time t that belong to the current iteration[1] and by W_t the total residual work for all packets in LA's buffer at time t. The next lemma provides a crucial observation: any packet accepted and processed by OPT while LA's buffer is full reduces total LA work by at least the maximal packet size in LA's buffer.

Lemma 3. *For any packet accepted by* OPT *at time $t \in [t_{\mathrm{con}}, t_{\mathrm{end}}]$ and processed by* OPT *during $[t_s, t_e] \subseteq [t_{\mathrm{con}}, t_{\mathrm{end}}]$, if $|\,\mathrm{IB}_{t-1}^{\mathrm{LA}}\,| = B$ and $|\,\mathrm{IB}_{t-1}^{\mathrm{OPT}}\,| = 0$ then $W_{t_e} \le W_{t-1} - M_t$.*

Proof. If LA's buffer is full then each packet p which is accepted by OPT is either pushed out or rejected by LA. If p is pushed out, the total work W_{t-1} is reduced by $M_{t-1} - r_t(p)$ immediately. Moreover, after processing such p, $W_{t_e} \le W_{t-1} - (M_{t-1} - r_t(p)) - r_t(p) = W_{t-1} - M_{t-1}$. Otherwise, since p is rejected by LA, $r_t(p) \ge M_{t-1}$, and thus $W_{t_e} \le W_{t-1} - r_t(p) \le W_{t-1} - M_{t-1}$. □

[1]This qualification deals with a boundary case: during the last B time slots of an iteration, when LA is transmitting, we set $M_t = 1$ for $t \in [t_{\mathrm{end}} - B, t_{\mathrm{end}}]$ independent of the newly arriving packets that will be processed on the next iteration; at any other time, M_t is simply the maximum residual work among all packets.

Next we denote by $f(B,W)$ the maximal number of packets that OPT can accept and process during $[t,t'] \subseteq [t_{\mathrm{con}}, t_{\mathrm{end}}]$, where $W = W_{t-1}$. The next lemma is crucial for the proof; it shows that OPT cannot have more than a logarithmic (in k) gain over LPO during the congestion period.

Lemma 4. $f(B,W) \le \log_{B/(B-1)} \frac{W}{B} \le \log_{B/(B-1)} k$.

Proof. All packets LPO transmits it does at the end of an iteration, hence, if the buffer of LPO is full, it will remain full until $t_{\mathrm{end}} - B$. At any time t, $M_t \ge \frac{W_t}{B}$: the maximal required processing is no less than the average. By Lemma 3, for every packet p accepted by OPT at time t, the total work $W = W_{t-1}$ is reduced by M_t after OPT has processed p. Therefore, after OPT processes a packet at time t', $W_{t'}$ is at most $W\left(1 - \frac{1}{B}\right)$. Let n be the number of packets transmitted by OPT during $[t", t]$. After n transmission rounds, the residual number of processing cycles in LA's buffer is $W(1-1/B)^n$, and the least possible work is B (B packets of size 1 each), so we get the bound as a solution of $W(1-1/B)^n = B$. □

Now we are ready to prove Theorem 1.

Proof of Theorem 1. Consider an iteration I that begins at time t' and ends at time t. If LA's buffer is not congested during I, by Lemma 2(3) OPT cannot transmit more than $|\,\mathrm{IB}_t^{\mathrm{LA}}\,|$ packets, and we are done.

Suppose now that LA's buffer is congested during I. By Lemma 2(1), LA's buffer remains full until the end of the iteration. If OPT transmits less than B packets during I then we are done. Otherwise, consider the time of the first congestion t_{con} during I. By Lemma 2(3), during $[t_{\mathrm{beg}}, t_{\mathrm{con}})$ OPT can transmit at most $B-1$ packets. By Lemma 4, OPT processes at most $\log_{B/(B-1)} k$ packets that arrive during $[t_{\mathrm{con}}, t_{\mathrm{end}}]$, plus at most B packets that it had in its buffer at time t_{con}, and then OPT flushes out at most B packets at time t_{end}. Thus, the total number of packets transmitted by OPT over a congested iteration is at most $3B + \log_{B/(B-1)} k$, as needed. □

Next we show that this bound, though seemingly weak, is in fact tight for some processing orders (namely RFIFO and RevPQ). Note also that the same lower bound holds for FIFO with recycles (RFIFO) where fully processed packets are immediately transmitted; this improves upon a previous result in [7]. We denote by LRFIFO and LRevPQ the respective lazy versions of RFIFO and RevPQ, as defined in Section 2. In Theorem 5 and all subsequent proofs of lower bounds, we prove the bound by presenting a hard instance of an input sequence on which the algorithm in question underperforms in comparison with OPT.

Theorem 5. *The competitive ratio of* LRFIFO*,* LRevPQ*, and* RFIFO *is at least* $(1 + \frac{1}{B} \log_{B/(B-1)} k)$.

Proof. We denote $\gamma = \frac{B-1}{B}$. At the first time slot (in the first burst), $(B-1) \times \boxed{k}$ packets arrive followed by $\boxed{\gamma k}$; OPT drops all $\boxed{k}$s and only leaves $\boxed{\gamma k}$. After γk steps, all three online policies will have $B \times \boxed{\gamma k}$ in the buffer, and then $\boxed{\gamma^2 k}$ arrives. Repeating this sequence ($\boxed{\gamma^{i+1} k}$ arrives after γ^i more steps) until $\mathrm{IB}^{\mathrm{ALG}}$ (where ALG

denotes any of LRFIFO, LRevPQ, or RFIFO) consists of $\boxed{1}$'s, we get that OPT has processed $\log_{1/\gamma} k = \log_{B/(B-1)} k$ packets while LRevPQ (LRFIFO) has processed none. Then we add a new burst of $B \times \boxed{1}$ packets, obtaining the bound. □

A natural question arises: is there any processing order with competitive ratio much better than this? To show that the processing order during an iteration has significant impact on the resulting behaviour, we consider the lazy Semi-FIFO policy with PQ processing order during iteration (LPQ). In [7], it was shown that a PQ policy that immediately transmits fully processed packets is optimal. Note that Semi-FIFO LPQ policy does not suffer from starvation of "heavy" packets in contrast to PQ. However, it pays a price for its "laziness". The following theorem shows how high this price is, i.e., we show an (asymptotically tight) bound on LPQ's competitiveness.

Theorem 6. *1.* LPQ *is at most* 2*-competitive.*

2. LPQ *is at least* $\left(2 - \frac{1}{B}\left\lceil\frac{B}{k}\right\rceil\right)$*-competitive.*

Proof. 1. Since PQ is optimal, during an iteration OPT cannot transmit more packets than reside in the LPQ buffer at the end of an iteration. Therefore, by Lemma 2(2) LPQ is at most 2-competitive.

2. For $k \geq B$, consider two bursts of packets: first $B \times \boxed{k}$ and then $(k-1)B$ steps each of the form $(B-1) \times \boxed{1}$. After these two bursts, OPT has processed $2B-1$ packets, and LPQ has processed B packets, so we can repeat them to get the asymptotic bound. For $k < B$, in the same construction $\left\lceil\frac{B}{k}\right\rceil$ packets are left in OPT's queue after $(k-1)B$ processing steps.

□

In fact, we can use the construction from Theorem 6(2) to obtain a general lower bound for every Semi-FIFO policy.

Theorem 7. *Any greedy Semi-FIFO policy is at least* $\left(1 + \frac{m-1}{B}\right)$*-competitive for* $m = \min\{k, B\}$.

Proof. We begin with $\boxed{m}$ followed by $(m-1) \times \boxed{1}$. Any Semi-FIFO policy will process the first packet in line over the next $m-1$ steps (there is nothing else to process—any other packet would be immediately transmitted upon processing, which contradicts the FIFO transmission order), while OPT drops the first packet and processes the rest. Then, in the second burst we add $B \times \boxed{1}$, thus filling out both buffers with $\boxed{1}$'s and erasing the difference between them. This process can be repeated once both queues have processed these B packets, obtaining the necessary bound. □

4. Applications of Semi-FIFO Results to other Transmission Orders

In this section, we apply our results for Semi-FIFO policies to policies whose transmission order corresponds to processing order. This is applicable to all processing orders we have defined except RevPQ since during an iteration the Semi-FIFO policy obeys the desired processing order, and between iterations packets with a single processing cycle can be transmitted in any order.

4.1. Comparison between Lazy and Non-Lazy Policies

First, we compare the performance of lazy and non-lazy policies. It would appear that lazy policies are inferior since they do not send out packets as soon as they can, thus creating extra congestion. Interestingly, however, there are inputs for which lazy algorithms outperform non-lazy greedy push-out policies with the same processing order. For instance, it was shown in [4] that greedy push-out FIFO is not comparable in the worst case with LFIFO. Here, we present an interesting result that any lazy policy, even $LRevPQ$ (largest required work first during iteration), can outperform greedy push-out FIFO and RFIFO policies, a rather counterintuitive result.

Theorem 8. *Any lazy policy* LA *(including* LRevPQ*) is incomparable with either* FIFO *or* RFIFO *in the worst case for every* $k > 2$ *and* $B > 2$*.*

Proof. We present four examples that show that LA can be either weaker or stronger than either FIFO or RFIFO.

1. LA $>$ FIFO. First burst: $\boxed{1}$, then $\boxed{2}$, then $(B-2) \times \boxed{1}$, so that both buffers are full afterwards. Now any lazy policy has no choice but process $\boxed{2}$ on the first time slot, and FIFO sends out the first $\boxed{1}$, leaving $B-1$ packets in the queue. Then, a packet of size k arrives; the lazy policy drops it, and FIFO accepts it at the tail. Then, during the next $B-1$ steps, LA and FIFO have processed $B-1$ packets, but FIFO has a packet of size k at head of line (and one additional packet transmitted before). We now add $\min(k-1, B-1) \times \boxed{1}$, and over the next $\min(k-1, B-1)$ steps LA processes them while FIFO processes the head of line packet; since before the last burst FIFO has transmitted one packet more than LA, we restrict $\min(k-1, B-1) > 1$ to guarantee that LA will transmit at least one packet more during processing of the last burst; another flushing burst of $B \times \boxed{1}$ fills out both buffers and both algorithms transmit an additional B packets.

2. LA $>$ RFIFO. First burst: $\frac{B}{2} \times \boxed{1}$, then $\frac{B}{2} \times \boxed{2}$. After $\frac{B}{2}$ processing steps, any lazy policy has a buffer full of $\boxed{1}$s, and RFIFO has $\frac{B}{2}$ $\boxed{2}$s and has processed $\frac{B}{2}$ packets. At this time, a second burst arrives with $\frac{B}{2} \times \boxed{2}$ (LA drops them all, RFIFO accepts). After B more steps, LA has processed B packets and has an empty buffer, and RFIFO has still processed only $\frac{B}{2}$ packets and has buffer full of $\boxed{1}$'s. At this point, the third burst of $B \times \boxed{1}$ fills both buffers, and LA has finally processed $\frac{3}{2}$ times more than RFIFO.

3. FIFO $>$ LA. First burst: $B \times \boxed{2}$. After B processing steps, any lazy policy has a buffer full of $\boxed{1}$'s, and FIFO has processed $\frac{B}{2}$ packets. At this time, a second burst arrives with $\frac{B}{2} \times \boxed{2}$ (LA drops them all, FIFO accepts). After B more steps, both LA and FIFO have processed B packets, but FIFO has $\frac{B}{2} \times \boxed{2}$ in the buffer and LA has it empty. Over the next B steps, we add $\boxed{1}$ per time slot, so at the end LA has processed $2B$ packets and FIFO has processed $\frac{3}{2}B$ packets and its buffer is full of $\boxed{1}$'s. Now we add $B \times \boxed{1}$, flushing out the difference

and leaving LA $\frac{6}{5}$ times ahead of FIFO (we can get a better bound for larger k but that is beside the point now).

4. RFIFO $>$ LA. First burst is exactly like (2), $\frac{B}{2} \times \boxed{1}$ followed by $\frac{B}{2} \times \boxed{2}$. After $\frac{B}{2}$ processing steps, any lazy policy has a buffer full of $\boxed{1}$'s, and RFIFO has $\frac{B}{2}$ $\boxed{2}$'s and has processed $\frac{B}{2}$ packets, so we can flush the buffers with $B \times \boxed{1}$, securing the advantage of $\frac{3}{2}$ times for RFIFO over LA.

□

Although in the worst case lazy policies are usually incomparable to their non-lazy counterparts, we will see from simulations that, as one would expect, they are often inferior in practice.

4.2. Emulation of Lazy Behaviour

All lazy algorithms delay transmission until the end of an iteration. However, for some values of B and k this can become unacceptable. The good news is that at least for some orders there exist policies that can emulate the behaviour of lazy policies but transmit fully processed packets upon completion of processing. To this end, it suffices to maintain an additional counter that holds the current buffer occupancy of the emulated lazy algorithm. Note that such an algorithm is not greedy since it cannot accept packets while the buffer of the lazy algorithm it simulates is full. This simulation can be performed at least for LFIFO and LRFIFO lazy algorithms. Policies emulating lazy algorithms that do not delay fully processed packets until the end of an iteration form an additional class of policies with provable performance guarantees.

5. Impact of Additional Constraints on Push-out

In this section we consider several interesting constraints on admission control; in particular, we apply additional constraints on what can be pushed out of the buffer. These constraints are natural and come from the real properties of actual systems. Observe that in this section we consider an updated version of Definition 3.1 that pushes a packet out according to the constraints for the push-out mechanism as defined below.

5.1. Non-Push-Out Policies

In this part we consider non-push-out greedy work-conserving policies. In has been shown in [7, 4] that a non-push-out work-conserving greedy policy with RFIFO or FIFO processing orders respectively are k-competitive. Here, we present the same result for any greedy Semi-FIFO policy.

Theorem 9. *Any greedy non-push-out Semi-FIFO policy* NPO *is at least* $\frac{k+1}{2}$*-competitive. Any lazy greedy non-push-out policy* NLPO *is at least* $(k-1)$*-competitive.*

Proof. The first burst begins with $B \times \boxed{k}$ that NPO has to accept and OPT drops. Then $\boxed{1}$s arrive over the next kB steps, so that after kB steps, NPO has a buffer full of $\boxed{1}$s and has processed B packets, while OPT has processed kB packets. Flushing the buffers with $B \times \boxed{1}$, we get the necessary bound. In the lazy case, we flush when NPO has a buffer full of $\boxed{1}$s and has not processed a single packet yet. □

5.2. Additive Constraints on the Push-Out Mechanism

In some systems it is problematic to push out a packet whose processing has already begun since potentially its state can be distributed internally in the network processor (for instance, on different cache levels). Therefore, we consider another class of policies that have restrictions on the push-out of already partially processed packets. Observe that for some orders this can lead to serious performance degradation. In such constrained RFIFO (FIFO with recycles) the processing order can be significantly worse than even RevPQ (maximal residual work first) processing order since after B processing cycles a policy with RFIFO processing order and congested buffer cannot push anything out any more. One of the processing orders where such a constraint is applied naturally is FIFO. Consider the 2RFIFO policy that emulates the RFIFO policy but already partially processed packets cannot be pushed out. The next theorem shows that such a small change in the push-out mechanism has significant impact on performance; we show an almost tight bound indicating that 2RFIFO is nearly k-competitive.

Theorem 10. *1.* 2RFIFO *is at least* $\frac{B(k-1)+\lfloor B/k \rfloor}{B}$*-competitive.*

2. 2RFIFO *is at most* k*-competitive.*

Proof. 1. For the lower bound, in the first burst we send $B \times \boxed{k}$. After B steps, 2RFIFO's buffer is full of $\boxed{k-1}$s and cannot push any of them out. Then, $\boxed{1}$s begin to arrive one by one over the next $B(k-1)$ time slots. After that, 2RFIFO has processed B packets while OPT has processed $B(k-1)$ packets; moreover, OPT could process an additional $\lfloor B/k \rfloor$ packets during the first B steps (this matters only if $B \geq k$).

2. For the upper bound, note that the trivial non-push-out upper bound of k applies to 2RFIFO as it does not preempt packets that it has spent processing time on, therefore, it cannot process less than $\lfloor T/k \rfloor$ packets in T non-idle processing cycles.

□

On the other hand, even a generally bad policy such as 2RFIFO can outperform the greedy push-out work-conserving FIFO policy.

Theorem 11. 2RFIFO *is incomparable with* FIFO *for every* $k > 2$ *and every* $B > 2$.

Proof. The example for 2RFIFO $>$ FIFO is the same as in Theorem 8(1); for FIFO $>$ 2RFIFO, see the example in Theorem 10. □

5.3. Multiplicative Constraints on the Push-Out Mechanism

In some systems we want to control the number of pushed out packets. This is done for at least two reasons: each admitted packet incurs some *copying cost* α, and pushing a packet out is not a simple operation. The addition of a copying cost α covers both cases. As a result, [7] introduced the notion of copying cost in the performance of transmission algorithms. Namely, if for an input sequence σ an online algorithm accepts A

packets and transmits T packets, its transmitted value is $\max(0, T - \alpha A)$. So in the case of a continuous burst it is possible that the transmitted value of a push-out policy can go to zero. As a result, in extreme cases it is possible that for some types of traffic and non-zero copying costs, a non-push-out policy outperforms push-out policies that suffer from "dummy paid" push-outs, so the copying cost is an additional control on the number of pushed out packets. To implement such a control mechanism, Keslassy et al. [7] introduced the greedy push-out work-conserving policy PQ_β that processes a packet with minimal required work first and in the case of congestion such a policy pushes out only if a new arrival has at least β times less work then the maximal residual work in PQ_β buffer[2]. They proved that the competitive ratio of PQ_β is at most

$$\frac{(1 + \log_{\frac{\beta}{\beta-1}} (k/2) + 2\log_\beta k)(1-\alpha)}{1 - \alpha \log_\beta k}.$$

Although PQ_β suffers from starvation of heavy packets (similarly to PQ), for us PQ_β provides a reference point for the other β-push-out Semi-FIFO policies. Next we prove a general upper bound for any lazy β-push-out Semi-FIFO policy – one more application of the powerful accounting infrastructure provided by lazy algorithms. Since for some orders like FIFO and RFIFO there are non-lazy versions with similar performance guarantees, the result below is even more interesting.

We begin with a modification of Lemma 3.

Lemma 12. *For any packet accepted by* OPT *at time* t *and processed by* OPT *during* $[t_s, t_e]$, $t \le t_s \le t_e$, *if* $|\,\mathrm{IB}_{t-1}^{\mathrm{LA}_\beta}\,| = B$ *and* $|IB_{t-1}^{\mathrm{OPT}}| = 0$ *then* $W_{t_e} \le W_{t-1} - \frac{M_{t-1}}{\beta}$.

Proof. If LA_β buffer is full then each packet p that is accepted by OPT either pushes out another packet in LA's buffer or is rejected by LA_β. If p pushes out another packet the overall work W_{t-1} is reduced by $M_t - r_t(p)$. Moreover, after processing of such p, $W_{t_e} \le W_{t-1} - (M_{t-1} - r_t(p)) - r_t(p) = W_{t-1} - M_{t-1}$. Otherwise, since p is rejected by LA_β, $r_t(p) \ge \frac{M_{t-1}}{\beta}$ and thus, $W_{t_e} \le W_{t-1} - r_t(p) \le W_{t-1} - \frac{M_{t-1}}{\beta}$. □

Theorem 13. *For copying cost* $0 < \alpha < \frac{1}{\log_\beta k}$, *the competitive ratio of* LA_β *is at most*

$$\left(3 + \frac{1}{B} \log_{\frac{\beta B}{\beta B - 1}} k\right) \frac{1-\alpha}{1 - \alpha \log_\beta k}.$$

Proof. The proof for the non-congested case is identical to the proof of Theorem 1. The proof for the congested case is somewhat different. Still at any time the maximum number of processing cycles left on a packet is at least the average number number of processing cycles of packets in LA_β buffer. Let n be the number of packets transmitted by OPT during $[t", t]$. So after n transmissions the residual number of processing cycles in LA_β buffer is $W(1 - \frac{1}{\beta B})^n$. Since initially $W \le kB$, OPT can transmit at most $\log_{\beta B/(\beta B-1)} k$ packets during each iteration from the first congestion and at

[2]Note that non-push-out policies are actually a special case of β-push-out policies for $\beta \ge k$.

most $\log_{\beta B/(\beta B-1)} k + 2B$ packets in total during an iteration I. By Lemma 2(2), at most B more packets may be flushed out between two consecutive iterations.

Now let us incorporate the copying cost α. By definition OPT never pushes out an already accepted packet so it will pay α for each transmitted packet. On the other hand, any packet transmitted by LA_β can transitively push out at most $\log_\beta k$ packets in the worst case before it is transmitted since every new packet has processing requirement at least β times less than the previous one. □

A general lower bound for all β-preemptive policies is given in the following theorem; it matches the extra factor in Theorem 13 as compared to Theorem 1.

Theorem 14. *For copying cost* $0 < \alpha < \frac{1}{\log_\beta k}$, LA_β *is at least* $\frac{1-\alpha}{1-\alpha \log_\beta k}$*-competitive.*

Proof. Suppose that during the first time slot n batches of B packets each arrive as follows: $B \times \boxed{\beta^n}$, $B \times \boxed{\beta^{n-1}}$, …, $B \times \boxed{\beta}$. For β-preemptive policies, each next batch pushes out the previous one. Therefore, the overall gain by any β-preemptive algorithm after processing this sequence is $B(1 - \alpha n)$. On the other hand, OPT will gain $B(1 - \alpha)$. This implies a lower bound on any β-preemptive algorithm for the maximal possible $n = \log_\beta k$. □

Note that Theorems 13 and 14 make sense only for $0 < \alpha < \frac{1}{\log_\beta k}$; for larger values of α, the construction of Theorem 14 shows that for some input sequences a β-preemptive algorithm may have zero or even negative gain after taking copying cost into account, so the notion of competitive ratio degenerates.

6. Simulation Study

In this section, we conduct a simulation study in order to further explore the performance of Semi-FIFO policies and validate theoretical results described above.

To the best of our knowledge, publicly available traffic traces do not contain information on processing requirements for the network packets in the traces. Furthermore, such requirements would be difficult to extract since they depend on specific hardware and network processor configuration of individual network elements. Moreover, such traces do not provide any information about the time scale; in particular, it is unclear how long should a time slot last. This information is essential in our model in order to determine both the number of processing cycles per time slot and traffic burstiness. We therefore perform the main body of our simulations on synthetic traces generated with ON-OFF Markov modulated Poisson processes (MMPP).

For the case of copying cost $\alpha = 0$, we use the greedy push-out work-conserving policy PQ as optimal; the optimality of this policy has been shown in [7]. Since there exists an optimal algorithm that never pushes out packets, for the case of $\alpha > 0$ we use the same optimal algorithm and simply multiply the number of packets it transmits by $(1 - \alpha)$; this is the optimal algorithm now (and directly applied PQ may be no longer optimal).

Our traffic comes from 500 independent sources, each generated by an on-off Markov modulated Poisson process (MMPP) which we use to simulate bursty traffic.

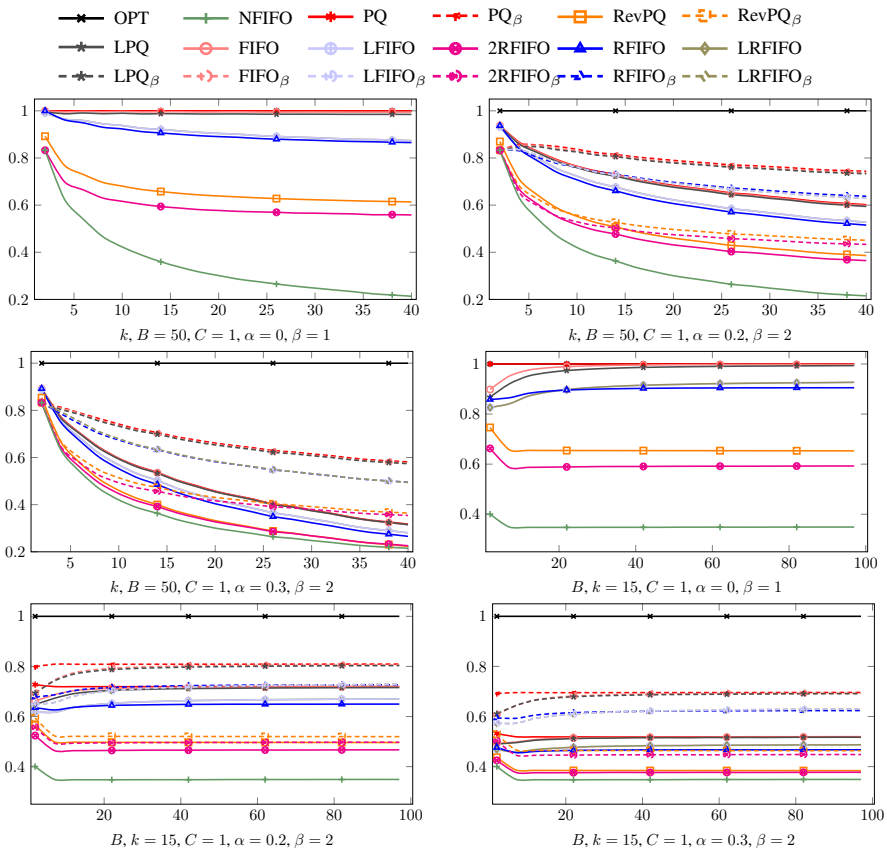

Figure 2: Simulation results. Solid lines show regular policies; dashed lines, their β-preemptive counterparts (for $\beta = 2$). Graphs, read top-down, left to right: (1) competitive ratio as a function of k for $B = 50$, $C = 1$, and $\alpha = 0$, (2) $\alpha = 0.2$, (3) $\alpha = 0.3$; (4) competitive ratio as a function of B for $k = 15$, $C = 1$, and $\alpha = 0$, (5) $\alpha = 0.2$, (6) $\alpha = 0.3$.

During every time slot, each source has probability 0.01 to be switched on, and once switched on, probability 0.2 to be switched back off. When a source is on, it emits packets with intensity λ_{on}, which represents one of the parameters governing traffic generation. Each generated packet is assigned required processing chosen uniformly at random from $\{1, \ldots, k\}$ (k being the maximum amount of processing required by any packet). We conducted our simulations for $5{,}000{,}000$ time slots and allowed different parameters to vary in each set of simulations. Our purpose has been to better understand the effect each parameter has on system performance and verify our analytic results.

By performing simulations for the maximal number of required passes k in the range $[1, 40]$, we evaluate the performance of our algorithms in settings ranging from underload to extreme overload, validating their performance in various traffic scenar-

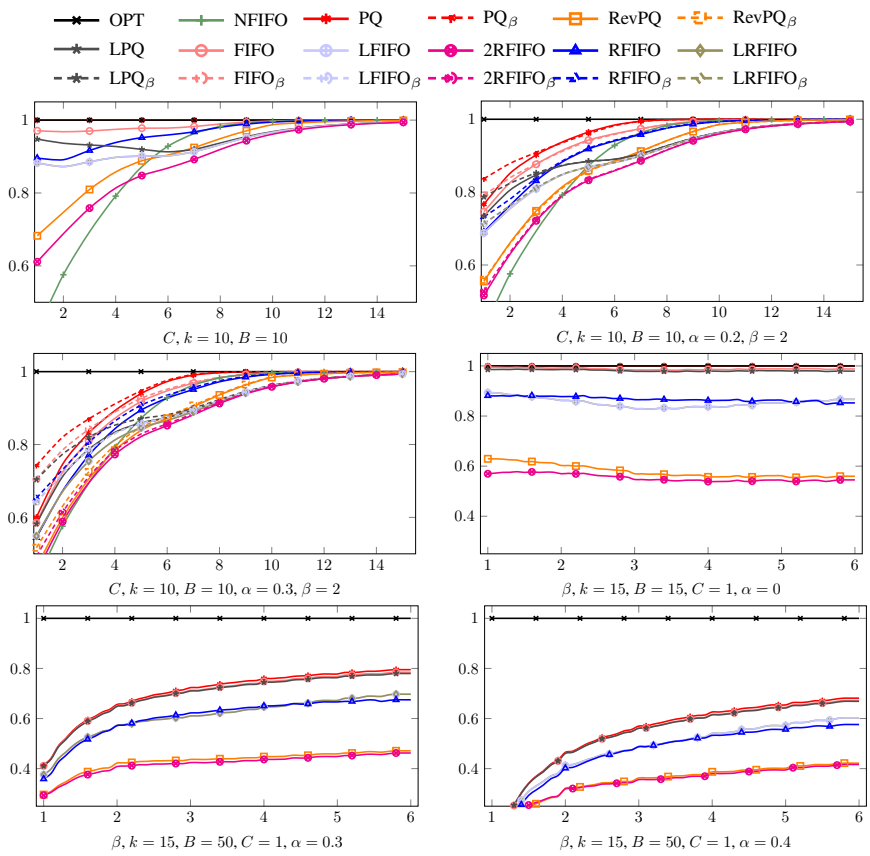

Figure 3: Simulation results. Solid lines show regular policies; dashed lines, their β-preemptive counterparts (for $\beta = 2$). Graphs, read top-down, left to right: (1) competitive ratio as a function of C for $k = 10$, $B = 10$, and $\alpha = 0$, (2) $\alpha = 0.2$, (3) $\alpha = 0.3$; (4) competitive ratio as a function of β for $k = 15$, $B = 50$, $C = 1$, and $\alpha = 0$, (5) $\alpha = 0.2$, (6) $\alpha = 0.3$.

ios. Figures 2 and 3 show simulation results. The vertical axis always represents the ratio between the algorithm's performance and OPT performance on the same arrival sequence (so the black line corresponding to OPT is always horizontal at 1). Throughout our simulation study, the standard deviation never exceeded 0.05 (deviation bars are omitted for readability). For every choice of parameters, we conducted 500,000 rounds (time slots) of simulation.

Figure 2 shows the results of our simulations. The Y-axis in all figures represents the ratio between an algorithm's performance and the optimal performance. We conduct four sets of simulations: in the first, we aim to better understand the dependence on the number of processing cycles, the second evaluates the dependency on the buffer size, the third explores the power of having multiple cores, while the fourth attempts to find optimal values of β. Our simulations are done for environments both with and

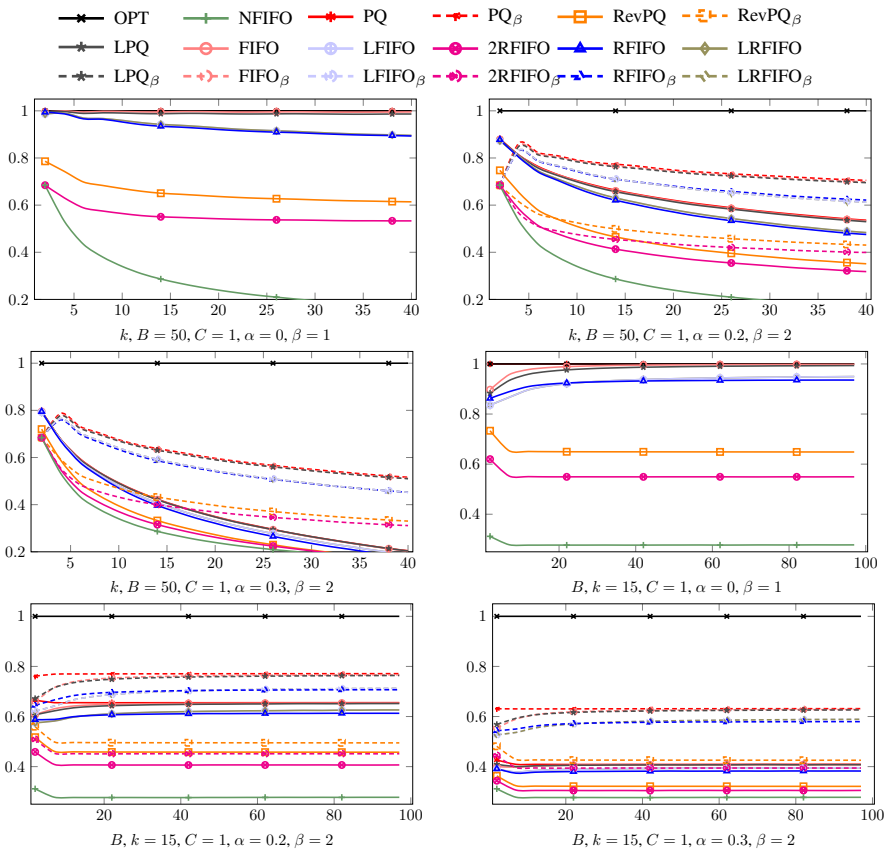

Figure 4: Simulation results on CAIDA traces. Solid lines show regular policies; dashed lines, their β-preemptive counterparts (for $\beta = 2$). Graphs, read top-down, left to right: (1) competitive ratio as a function of k for $B = 50$, $C = 1$, and $\alpha = 0$, (2) $\alpha = 0.2$, (3) $\alpha = 0.3$; (4) competitive ratio as a function of B for $k = 15$, $C = 1$, and $\alpha = 0$, (5) $\alpha = 0.2$, (6) $\alpha = 0.3$.

without coping cost. For the case of β-push-out algorithms, we also evaluate dependency on the value of β. One of the most interesting aspects arising from our simulation results is the fact that they seem to imply that our *worst-case* analysis has been beneficial for the design of algorithms, as the algorithms that perform well in the *worst-case* also work well *on average*.

6.1. Variable Number of Processing Cycles

For the first set of simulations, we restrict our attention to the single core case ($C = 1$) and consider fixed buffer size $B = 50$. The first three graphs on Fig. 2 show that OPT's advantage over the other algorithms grows as k increases. On the other hand, interestingly, as α increases β-push-out algorithms (for $\beta > 1$) begin to outperform "standard" push-out algorithms with $\beta = 1$. We show the graphs for the

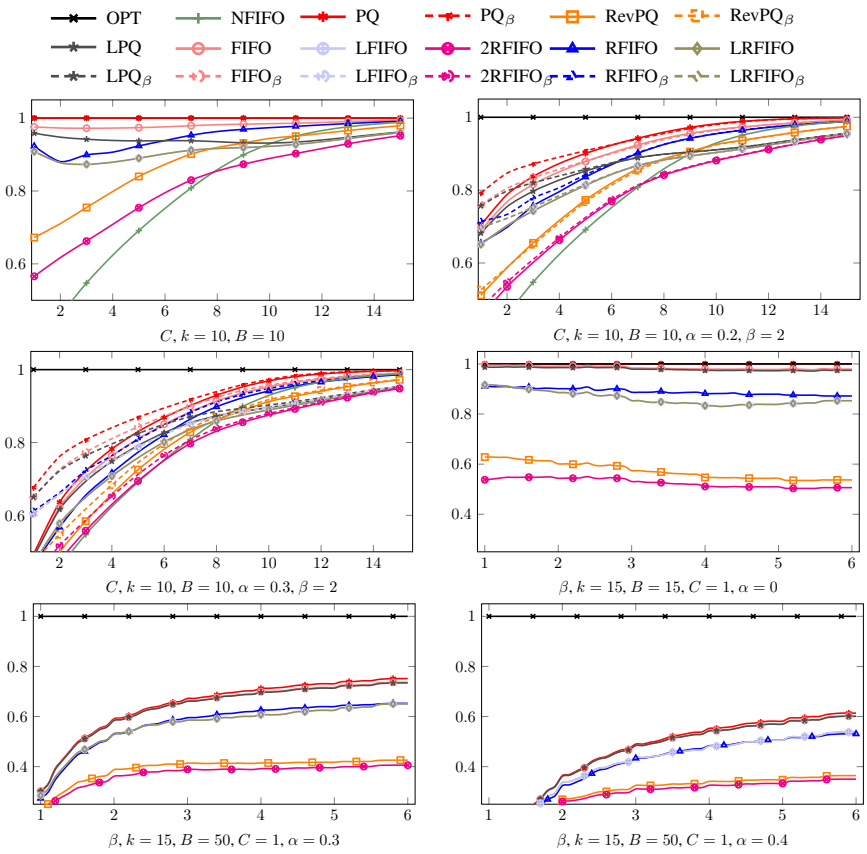

Figure 5: Simulation results on CAIDA traces. Solid lines show regular policies; dashed lines, their β-preemptive counterparts (for $\beta = 2$). Graphs, read top-down, left to right: (1) competitive ratio as a function of C for $k = 10$, $B = 10$, and $\alpha = 0$, (2) $\alpha = 0.2$, (3) $\alpha = 0.3$; (4) competitive ratio as a function of β for $k = 15$, $B = 50$, $C = 1$, and $\alpha = 0$, (5) $\alpha = 0.2$, (6) $\alpha = 0.3$.

value $\beta = 2$.

6.2. Variable Buffer Size

In this set of simulations we evaluate the performance of our algorithms for different values of B, $1 \leq B \leq 100$. We restrict our attention to $C = 1$ and $k = 15$ and evaluate different values of B. Graphs four through six on Fig. 2 present our results. Unsurprisingly, performance of FIFO-based algorithms significantly improves as the buffer size increases; the difference between OPT and other push-out algorithms visibly reduces. Non-push-out policies, however, still lose since a much bigger buffer is required to avoid congestion altogether. As above, 2-push-out algorithms begin to outperform their 1-push-out counterparts when α grows to 0.2 and more.

6.3. Variable Number of Cores

In this set of simulations, we evaluate the performance of our algorithms for variable values of C, $1 \leq C \leq 15$ (leaving the average arrival rate constant). The first three graphs on Fig. 3 present our results; the performance of all algorithms, naturally, improves drastically as the number of cores increases (even non-push-out). Interestingly, push-out capability becomes less important since buffers are congested less often. Moreover, at some point non-push-out algorithms begin to outperform lazy algorithms that still pay for lazy behaviour.

6.4. Variable Value of β

In this set of simulations we evaluate the performance of β-push-out algorithms for fixed $C = 1$, $k = 15$, and $B = 15$ for different values of $\beta \in [1, 6]$ (the last three graphs on Fig. 3). The difference between β-push-out algorithms and OPT remains approximately the same as β grows.

6.5. Experiments on CAIDA traces

Although the simulation results above are convincing, they do rely completely upon a synthetic scheme. To validate our synthetic generation, we have used CAIDA traces available at [33] that contain timestamps of arriving packets. Instead of relying on MMPP generation for the number of packets, we have now chosen a time interval to serve as a time slot and took the number of packets from the trace. Unfortunately, as we have already mentioned, publicly available traces do not contain information regarding required processing, so the required processing of the packets was still generated synthetically as above.

Figures 4 and 5 mirror Figures 2 and 3: they depict the same experiments, but the number of packets is now generated by a CAIDA trace. It is clear from the figures that the plots look almost exactly the same as in completely synthetic simulations, with very minor differences; we thus conclude that the synthetic study is as representative of the real situation as we can get without additional information on required processing.

7. Discussion

Since packet processing engines (PPE)s based on specialized ASICs are an order of magnitude faster than network processors and two orders of magnitude faster than general purpose CPUs from the same generation, new PPEs based on specialized ASICs still will be introduced in future systems [34]. There is a fundamental tradeoff between expressiveness of PPEs and its cost. Though PPEs based on general purpose CPUs are more flexible, standardization efforts of southbound API as OpenFlow can make them similarly expressive as PPEs based in specialized ASICSs. In these settings understanding what should be flexible becomes extremely important; as a result heterogeneous traffic characteristics together with different objectives pose design challenges to novel architectures and management policies.

In this work, we have studied the impact of processing order and additional constraints on push-out mechanism of admission control on performance of different buffer

Algorithm/family	Lower bound	Upper bound
Semi-FIFO	$1+\frac{\min\{k,B\}-1}{B}$	open problem
Lazy	$1+\frac{\min\{k,B\}-1}{B}$	$3+\frac{1}{B}\log_{\frac{B}{B-1}} k$
LRFIFO, LRevFIFO	$1+\frac{1}{B}\log_{\frac{B}{B-1}} k$	$3+\frac{1}{B}\log_{\frac{B}{B-1}} k$
LPQ	$2-\frac{1}{B}\left\lceil\frac{B}{k}\right\rceil$	2
2RFIFO	$k-1+\frac{1}{B}\left\lfloor\frac{B}{k}\right\rfloor$	k
Semi-FIFO β-preemptive	$\frac{1-\alpha}{1-\alpha\log_\beta k}$	open problem
Lazy β-preemptive	$\frac{1-\alpha}{1-\alpha\log_\beta k}$	$\left(3+\frac{1}{B}\log_{\frac{\beta B}{\beta B-1}} k\right)\frac{(1-\alpha)}{1-\alpha\log_\beta k}$
Non-push-out	$\frac{k+1}{2}$	k
Lazy non-push-out	$k-1$	k

Table 1: Results summary: lower and upper bounds.

management algorithms. For this purpose, we have decoupled processing and transmission orders and built a taxonomy of Semi-FIFO policies. For the newly introduced classes of policies, we have proven worst-case performance guarantees and provided lower bounds. Table 1 summarizes the lower and upper bounds for the policies we have introduced together with some already known algorithms. In addition, we have compared different classes of algorithms and provided incomparability results.

Further work may include improving the bounds shown in Table 1 and extending this research to other situations: multiple queues in a switch, randomized policies, learning policies, consideration of goodput instead of throughput and other extensions and refinements.

References

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

[2] T. Wolf, P. Pappu, M. A. Franklin, Predictive scheduling of network processors, Computer Networks 41 (5) (2003) 601–621.

[3] K. Kogan, A. López-Ortiz, S. I. Nikolenko, G. Scalosub, M. Segal, Large profits or fast gains: A dilemma in maximizing throughput with applications to network processors, J. Network and Computer Applications 74 (2016) 31–43.

[4] K. Kogan, A. López-Ortiz, S. I. Nikolenko, A. V. Sirotkin, Online scheduling FIFO policies with admission and push-out, Theory Comput. Syst. 58 (2) (2016) 322–344.

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

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

[7] I. Keslassy, K. Kogan, G. Scalosub, M. Segal, Providing performance guarantees in multipass network processors, in: INFOCOM, 2011, pp. 3191–3199.

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

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

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

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

[12] W. Aiello, Y. Mansour, S. Rajagopolan, A. Rosén, Competitive queue policies for differentiated services, J. Algorithms 55 (2) (2005) 113–141.

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

[14] Y. Mansour, B. Patt-Shamir, D. Rawitz, Overflow management with multipart packets, in: INFOCOM, 2011, pp. 2606–2614.

[15] S. Albers, M. Schmidt, On the performance of greedy algorithms in packet buffering, SIAM Journal on Computing 35 (2) (2005) 278–304.

[16] Y. Azar, Y. Richter, An improved algorithm for CIOQ switches, ACM Transactions on algorithms 2 (2) (2006) 282–295.

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

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

[19] 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.

[20] P. Eugster, K. Kogan, S. I. Nikolenko, A. V. Sirotkin, Heterogeneous packet processing in shared memory buffers, J. Parallel Distrib. Comput. 99 (2017) 1–13.

[21] P. T. Eugster, A. Kesselman, K. Kogan, S. I. Nikolenko, A. Sirotkin, Essential traffic parameters for shared memory switch performance, in: SIROCCO, 2015, pp. 61–75.

[22] S. Albers, T. Jacobs, An experimental study of new and known online packet buffering algorithms, Algorithmica 57 (4) (2010) 725–746.

[23] Z. Zhang, F. Li, S. Chen, Online learning approaches in maximizing weighted throughput, in: IPCCC, 2010, pp. 206–213.

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

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

[26] A. Davydow, P. Chuprikov, S. I. Nikolenko, K. Kogan, Throughput optimization with latency constraints, in: INFOCOM, 2017, pp. 1–9.

[27] L. Schrage, A proof of the optimality of the shortest remaining processing time discipline, Operations Research 16 (1968) 687–690.

[28] S. Leonardi, D. Raz, Approximating total flow time on parallel machines, in: STOC, 1997, pp. 110–119.

[29] S. Muthukrishnan, R. Rajaraman, A. Shaheen, J. E. Gehrke, Online scheduling to minimize average stretch, SIAM Journal on Computing 34 (2) (2005) 433–452.

[30] R. Motwani, S. Phillips, E. Torng, Non-clairvoyant scheduling, Theoretical Computer Science 130 (1) (1994) 17–47.

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

[32] 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.

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

[34] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. A. Mujica, M. Horowitz, Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn, in: SIGCOMM, 2013, pp. 99–110.