Theory Comput Syst (2016) 58:322–344
DOI 10.1007/s00224-015-9626-4

Online Scheduling FIFO Policies with Admission and Push-Out

Kirill Kogan[1] · Alejandro López-Ortiz[2] · Sergey I. Nikolenko[3,4] · Alexander V. Sirotkin[5,6]

Published online: 12 April 2015

Abstract We consider the problem of managing a bounded size First-In-First-Out (FIFO) queue buffer, where each incoming unit-sized packet requires several rounds of processing before it can be transmitted out. Our objective is to maximize the

A preliminary version of this paper has appeared in [21]; compared to the conference version, this paper contains new results about the two-valued case, changes our constructions so that they more strictly adhere to the FIFO processing order, and presents a modified proof of the main theorem that improves over previous results.

✉ S. I. Nikolenko
sergey@logic.pdmi.ras.ru

K. Kogan
kirill.kogan@imdea.org

A. López-Ortiz
alopez-o@uwaterloo.ca

A. V. Sirotkin
alexander.sirotkin@gmail.com

1 IMDEA Networks Institute, Madrid, Spain

2 School of Computer Science, University of Waterloo, Waterloo, Canada

3 Laboratory for Internet Studies, National Research University Higher School of Economics, St. Petersburg, Russian Federation

4 Laboratory of Mathematical Logic, Steklov Mathematical Institute at St. Petersburg, St. Petersburg, Russian Federation

5 International Laboratory for Applied Network Research, National Research University Higher School of Economics, Moscow, Russia

6 Theoretical and Interdisciplinary Computer Science Laboratory, St. Petersburg Institute for Informatics and Automation of the RAS, St. Petersburg, Russia

total number of successfully transmitted packets. We consider both push-out (when a policy is permitted to drop already admitted packets) and non-push-out cases. We provide worst-case guarantees for the throughput performance of our algorithms, proving both lower and upper bounds on their competitive ratio against the optimal algorithm, and conduct a comprehensive simulation study that experimentally validates predicted theoretical behavior.

Keywords Scheduling · Buffer management · First-in-first-out queueing · Switches · Online algorithms · Competitive analysis

1 Introduction

Modern day computing faces increasingly heterogeneous tasks, varying in importance, size, processing requirements, and other characteristics. Such scenarios are encountered, for example, in OS caching environments, job-shop scheduling, and, most notably, network processors dealing with packet-switched traffic. In many of these environments the queue might be constrained to use a limited size buffer, which further restricts the ability to provide performance adequate to the underlying computing tasks.

In this work we consider the problem of scheduling packets with heterogeneous required processing in case when admission control and push out may be allowed and the queue is bounded, i.e. no more than B unit-sized packets can be waiting at any given time. This scenario is motivated by real life applications in buffers on network processors (NP).

In what follows we adopt the terminology used to describe buffer management problems and focus our attention on the general model where we are required to manage admission control and scheduling units of a single bounded size queue that processes and transmits packets in First-In-First-Out (FIFO) order. Because it is simple and does not require packet reordering at the final destination, FIFO processing has become a *de facto* standard in modern packet processors [7, 9, 12]. We consider a model where arriving traffic consists of unit-sized *packets*, and each packet has a *processing requirement* (at most k in processor cycles). Per-packet processing requirements are often highly regular and predictable for a fixed configuration of network elements and well-defined as a function of the features associated with the "flow" and the network element configuration [34]. A packet is successfully *transmitted* only after the scheduling unit has scheduled the packet for processing for at least its required number of cycles. If a packet is dropped upon arrival or pushed out from the queue after being admitted due to admission control policy considerations (if push-out is allowed), the packet is lost without gain to the algorithm's throughput. Our objective is to maximize the total number of successfully transmitted packets.

1.1 Our Contributions

For online settings, we propose algorithms with provable performance guarantees. We consider both push-out (when the algorithm can drop a packet from the queue)

and non-push-out cases. We show that the competitive ratio obtained by our algorithms depends on the maximum number k of processing cycles required by a packet, but none of our algorithms need to know the maximum number of processing cycles k in advance.

We discuss the non-push-out case in Section 2 and show that the on-line non-push-out greedy algorithm NPO (Algorithm 1) is k-competitive, and that this bound is tight.

For the push-out case, we consider a simple greedy algorithm PO (Algorithm 2), which in the case of congestion pushes out the first packet with maximal required processing, and introduce the Lazy-Push-Out algorithm (LPO, Algorithm 3). LPO works similar to PO in push-out decisions but does not transmit packets if there still exist packets in the queue that still have to be processed (i.e., LPO waits until all packets in the buffer have zero cycles left and then sends C "fully processed" packets per time slot, where C is the number of cores). Note that LPO preserves FIFO processing order and postpones transmission of "fully processed" packets only for simplicity of description. In practice, we can define a new online algorithm with the same performance that will emulate the buffer occupancy of LPO and will not artificially delay the transmission of "fully processed" packets.

Intuitively, it seems that PO should outperform LPO since PO tends to empty its buffer faster but we demonstrate that these algorithms are "incomparable" in the worst case. Although we provide a lower bound for PO, the main result of this work is an upper bound on the competitiveness of LPO. Namely, we demonstrate that LPO is at most $(\max\{1, \ln k\} + 2 + o(1))$-competitive. We also prove several lower bounds on the competitiveness of both PO and LPO for different values of B and k. For a special case of systems with only two different required processing values, 1 and k (this is a special case that often occurs in practice), our bound improves to $\left(2 + \frac{1}{B}\right)$. These results are presented in Section 3.

The competitiveness result for LPO is interesting in itself since "lazy" algorithms provide a well-defined accounting infrastructure; we hope that a similar approach can be applied to other systems in similar settings. In Section 4, we conduct a comprehensive simulation study to experimentally verify the performance of the proposed algorithms. Section 5 concludes the paper.

1.2 Related Work

Keslassy et al. [15] and later [8, 11, 20, 22, 23] considered buffer management for packets with heterogeneous processing requirements in various settings. They study both SRPT (shortest remaining processing time) and FIFO (first-in-first-out) 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 (FIFO or SRPT). They showed competitive algorithms and worst-case lower bounds.

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., a recent survey by Goldwasser [13] and later Nikolenko et al. [29] provide

an excellent overview of this field). This line of research, initiated in [19, 26], has received tremendous attention in the past decade.

Various models have been proposed and studied, including, among others, QoS-oriented models where packets have weights [1, 10, 14, 18, 19, 26]. A related field that has received much attention in recent years focuses on various switch architectures and aims at designing approximation algorithms for such scenarios; see, e.g., [2, 3, 16, 17, 24].

There is a long history of OS scheduling for multithreaded processors which is relevant to our research. For instance, the SRPT algorithm has been studied extensively in such systems, and it is well known to be optimal with respect to the mean response [32]. Additional objectives, models, and algorithms have been studied extensively in this context [25, 27, 28]. For a comprehensive overview of competitive online scheduling for server systems, see a survey by Pruhs [30]. When comparing this body of research with our proposed framework, one should note that OS scheduling is mostly concerned with average response time, but we focus on estimation of 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.

The model considered in our work is also closely related to job-shop scheduling problems [5], most notably hybrid flow-shop scheduling [31] in scenarios where machines have bounded buffers and cannot drop or push out packets.

1.3 Model Description

We consider a buffer with 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 motivation of why such information may be available see [34]. Although the value of k will play a fundamental role in our analysis, we note that our algorithms need not know k in advance. Note that for $k = 1$ the model trivializes to a single queue of uniform packets.

In what follows, we adopt the terminology used in [23]. The queue performs two main tasks, namely *buffer management*, which handles admission control of newly arrived packets and push-out of currently stored packets, and *scheduling*, which decides which of the currently stored packets will be scheduled for processing. The scheduling will be determined by the FIFO order employed by the queue. Our framework assumes a multi-core environment, where we have C processors, and at most C packets may be chosen for processing at any given time.[1] This simple setting suffices to show both the difficulties of the model and our algorithmic scheme.

We assume discrete slotted time. Each time slot t consists of three phases:

(i) *arrival*: new packets arrive, and the buffer management unit performs admission control and, possibly, push-out;

[1] Note that two cores are not allowed to process the same packet simultaneously.

(ii) *processing*: C packets (if exist) are selected for processing by the scheduling module; the packets remain at their places after processing and are not recycled to the back of the queue;
(iii) *transmission*: C packets (if exist) with zero remaining processing can be transmitted and leave the queue.

In the remainder of this paper we assume the system selects a single packet for processing at any given time (i.e., $C = 1$) in the theorems; however, variable C will resurface in the simulations (Section 4).

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 overall number of packets transmitted from the queue.

For lazy algorithms, packets with zero remaining processing cycles can be delayed. But even in this case processing and transmission order follows arrival order, so the FIFO property among all processed and transmitted packets is satisfied.

The FIFO property is important in many networking applications. However, the term "FIFO" has been slightly ambiguous in our model so far; it can be understood to mean either

(i) "transmission order coincides with arrival order" or
(ii) "both transmission and processing order coincide with arrival order".

The difference is that in the former case, a policy is free to choose any packet it wishes for processing and only has to transmit the processed packets in the arrival order; in [20], such policies are called *semi-FIFO*, and a comprehensive treatment of such policies is presented. In this work, all considered policies conform to the latter, more strict understanding, so processing order also must follow arrival order.

We define a *greedy* buffer management policy as a policy that accepts all arrivals if there is available buffer space in the queue. A policy is *work-conserving* if it always processes whenever its buffer is nonempty. We say that an arriving packet p *pushes out* a packet q that has already been accepted into the buffer when q is dropped in order to free buffer space for p, and p is admitted to the buffer instead in FIFO order. A buffer management policy is called a *push-out* policy whenever it allows packets to push out currently stored packets. Figure 1 shows a sample time slot in our model (for the greedy push-out case). For an algorithm ALG and time slot t, we denote by IB_t^{ALG} the set of packets stored in ALG's buffer at time slot t after arrival but before transmission (see Fig. 1).

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, its number of *residual processing cycles*, denoted $r_t(p)$, is defined to be the number of processing cycles it requires before it can be successfully transmitted.

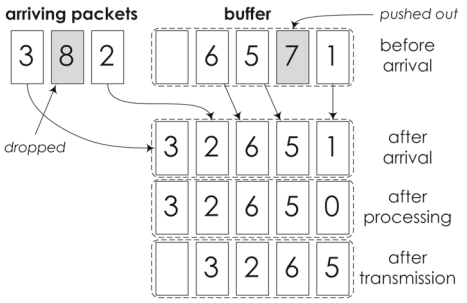

Fig. 1 Zoom in on a single time slot for a greedy push-out work-conserving algorithm. In this slot, $\text{IB}^{\text{ALG}} = \{\boxed{3}, \boxed{2}, \boxed{6}, \boxed{5}, \boxed{1}\}$

In this work we do not assume any specific traffic distribution but rather analyze our switching policies against adversarial traffic using competitive analysis [4, 33], which provides a uniform throughput guarantee for all traffic patterns. An online algorithm A is said to be $\alpha - competitive$ (for some $\alpha \geq 1$) if for any arrival sequence σ the number of packets successfully transmitted by A is at least $1/\alpha$ times the number of packets successfully transmitted by an optimal solution (denoted OPT) obtained by an offline clairvoyant algorithm. Also two algorithms ALG_1 and ALG_2 are *incomparable in the worst case* if there are two inputs I_1 and I_2, such that ALG_1 outperforms ALG_2 on the input I_1 and ALG_2 outperforms ALG_1 on the input I_2; otherwise, ALG_1 and ALG_2 are comparable in the worst case.

1.4 Proposed Algorithms

In this work, we study both push-out and non-push-out algorithms. The Non-Push-Out Algorithm (NPO) is a simple greedy work-conserving policy that accepts a packet if there is enough available buffer space. Already admitted packets are processed in First-In-First-Out order (head of line packet is always selected for processing). If during arrivals NPO's buffer is full then any arriving packet is dropped even if it has less processing required than a packet already admitted to NPO's buffer (see Algorithm 1).

Algorithm 1 NPO(p): Arrival Phase

```
1: if buffer occupancy is less than B then
2:     accept p at the tail of the queue
3: else
4:     drop p
```

Next we introduce two push-out algorithms. The Push-Out Algorithm (PO) is also greedy and work-conserving, but now, when an arriving packet p requires less processing cycles than at least one packet in its buffer, PO pushes out the first packet with the maximal number of processing cycles in its buffer and accepts p according to FIFO order (see Algorithm 2). For processing, PO always selects the first packet in the queue.

Algorithm 2 PO(p): Arrival Phase

```
1: if buffer occupancy is less than B then
2:     accept p
3: else
4:     let q be the first packet with maximal number of residual processing
5:     if r_t(p) < r_t(q) then
6:         drop q and accept p at the tail of the queue
```

The second algorithm is a new Lazy-Push-Out algorithm LPO that mimics the behaviour of PO with two important differences (see Algorithm 3):

(i) LPO does not transmit a head of line packet with zero processing cycles left if still has packets with residual processing left in the buffer, until the buffer contains only packets with zero residual processing cycles;
(ii) once all packets in LPO's buffer (say there are m packets there) have zero processing cycles remaining, LPO transmits them over the next m processing cycles.[2]

LPO has push-out similar to PO: if an arriving packet p requires less processing than the first packet q with maximal number of processing cycles in LPO's buffer, p pushes out q. Note that processing order does follow arrival order for LPO, it is only that we sometimes forgo the possibility to transmit an already processed packet.

Algorithm 3 LPO(p): Buffer Management Policy

Arrival Phase:

```
1: if p is the first packet to arrive then
2:     ToBeTransmitted := 0                          ▷ iteration begins
3: PO(p)                                             ▷ call to arrival phase of PO
```

Processing Phase:

```
1: Process the first from HOL packet p with r(p) > 0
2: if all packets p in the buffer have r(p) = 0 then
3:     ToBeTransmitted := # of packets in the queue  ▷ iteration ends
```

Transmission Phase:

```
1: if ToBeTransmitted > 0 then
2:     transmit HOL packet                           ▷ between two consecutive iterations
3:     ToBeTransmitted := ToBeTransmitted − 1
4: else
5:     ToBeTransmitted := 0                          ▷ new iteration begins
```

Intuitively, LPO is a weakened version of PO since PO tends to empty its buffer faster. Simulations also support this view (see Section 4). However, the follow-

[2]Here, again, there is room for different possible interpretations of LPO; we might assume that packets with zero processing left can be transmitted all at once. This would make LPO better and improve the upper bound in Theorem 5 by 1. Since our main results deal with upper bounds, we make the least favourable choice and assume that only one packet per time slot can be transmitted (C packets in the multicore case).

ing theorem shows that LPO and PO are incomparable in the worst case, i.e., none of them dominates the other on all inputs. To illustrate the proposed algorithms once again, hard examples in the proof of Theorem 1 are also illustrated on Fig. 2.

Theorem 1 (1) *There exists a sequence of inputs on which* PO *processes* ≥ 2 *times more packets than* LPO.

(2) *There exists a sequence of inputs on which* LPO *processes* $\geq \frac{3}{2+3/k}$ *times more packets than PO.*

Proof To see (1), consider two bursts of B packets:

(i) a burst of B packets with required processing 1 arriving at time slot $t = 1$ and
(ii) a burst of B packets with required processing 1 arriving at time slot $t = B$.

By the time the second burst arrives, LPO has processed no packets and has B packets with required processing 0 in its buffer, while PO has processed B packets. Then both algorithms transmit B packets over the next B time slots. Since we have arrived at a state where both algorithms have empty buffers, we can repeat the procedure, getting an asymptotic bound.

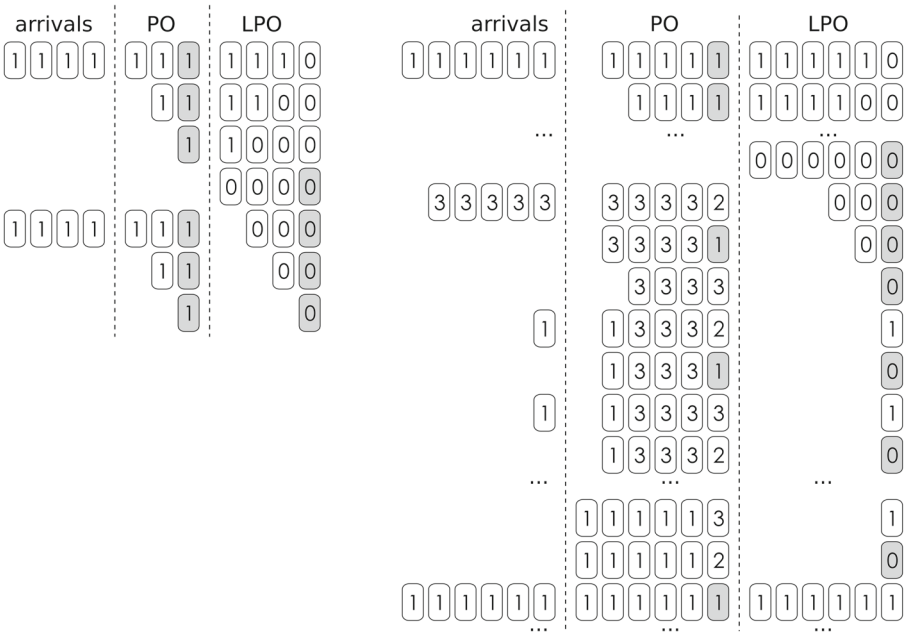

Fig. 2 Two packet processing examples from Theorem 1. Time flows from top to bottom, buffer states for each timeslot are shown after the transmission phase. Packets that will be transmitted out on the next time slot are shown in grey. Up: case (1), PO processes more packets than LPO. Right: case (2), LPO processes more packets than PO

To prove (2), suppose that $k < \frac{B}{3}$. The following table demonstrates the sequence of arrivals and the execution of both algorithms (#ALG denotes the number of packets processed by ALG up to this time; no packets arrive during time slots not shown in the table).

t	Arriving	$\mathrm{IB}_t^{\mathrm{LPO}}$	# LPO	$\mathrm{IB}_t^{\mathrm{PO}}$	# PO
1	$1 \times B$	$1 \ldots 1$	0	$1 \ldots 1$	0
B	$k \times \frac{3B}{k}$	$0 \ldots 0$	0	$k \ldots k$	B
$2B$	1	1	B	$1\ k \ldots k$	$B\left(1+\frac{1}{k}\right)$
$2B+2$	1	1	$B+1$	$1\ 1\ k \ldots k-2$	$B\left(1+\frac{1}{k}\right)$
$\ldots$		$\ldots$		$\ldots$	
$4B-1$	none	0	$2B-1$	$1 \ldots 1\ 1$	$B\left(1+\frac{3}{k}\right)-1$
$4B$	$1 \times B$	$1 \ldots 1$	$2B$	$1 \ldots 1$	$B\left(1+\frac{3}{k}\right)$
$6B$	none	$\emptyset$	$3B$	$\emptyset$	$B\left(2+\frac{3}{k}\right)$

Similar to (1), we can repeat this sequence. □

LPO is an online push-out algorithm that obeys the FIFO ordering model, so its competitiveness is an interesting result by itself. But we believe this type of algorithms to be a rather promising direction for further study since they provide a well-defined accounting infrastructure that can be used for system analysis in different settings. From an implementation point of view we can define a new on-line algorithm that will emulate the behaviour of LPO but will not delay the transmission of processed packets. Observe that such an algorithm is not greedy. Although we will briefly discuss the competitiveness of an NPO policy and lower bounds for PO, in what follows NPO and PO will be mostly used as a reference for the simulation study.

2 Competitiveness of the Non-Push-Out Policy

The following theorem provides a tight bound on the worst-case performance of NPO.

Theorem 2

(1) *For a sufficiently long arrival sequence, the competitiveness of* NPO *is at least* k.
(2) *For a sufficiently long arrival sequence, the competitiveness of* NPO *is at most* k.

Proof

(*1*) *Lower bound*. Consider the following set of arrivals: during the first time slot, there arrive $B \times \boxed{k}$ thus, over the next k time slots the buffer of NPO is full. During this time interval, there arrives a $\boxed{1}$ on each time slot, then $1 \times \boxed{k}$, and then again $\boxed{1}$ every turn for the next k slots, and so on. During each iteration, OPT transmits k packets while NPO transmits only one packet.

(*2*) *Upper bound*. Observe that NPO must fill up its buffer before it drops any packets. Moreover, as long as NPO's buffer is not empty after at most k time steps NPO must transmit its HOL packet. This means that NPO is transmitting at a rate of at least one packet every k time steps, while OPT in the same time interval transmits at most k packets. Hence, the number of transmitted packets at time t for NPO is at least $\lfloor t/k \rfloor$ while OPT transmits at most t packets for a competitive ratio of k over the period when NPO has nonempty buffer.

If, on the other hand, NPO empties its buffer first, it means that there were no packet arrivals since the NPO buffer went below the $B - 1$ threshold at a time t. From that moment on NPO empties its buffer transmitting thus at least $B - 1$ packets, while OPT transmits at most B packets.

So in total the number of packets transmitted by NPO is at least $\lfloor \frac{t}{k} \rfloor + B - 1$ while the total number of packets transmitted by OPT is $t + B$. Thus, for sufficiently long input sequences NPO is k-competitive. □

As demonstrated by the above results, the simplicity of non-push-out greedy policies does have its price. In the following sections we explore the benefits of introducing push-out policies and provide an analysis of their performance.

3 Competitiveness of Push-Out Policies

In this section, we show lower bounds on the competitive ratio of PO and LPO algorithms (Section 3.1) and prove an upper bound for LPO (Section 3.2); in Section 3.3, we consider a special case of packets of two sizes (a special case important in practice) and prove a tight pair of lower and upper bounds for this case.

3.1 Lower bounds

In this part we consider lower bounds on the competitive ratio of PO and LPO for different values of k and B.

Theorem 3 *For $k \geq B$, the competitive ratio of* PO *is at least* $2\left(1 - \frac{1}{B}\right)$, *and the competitive ratio of* LPO *is at least* $3\left(1 - \frac{1}{B}\right)$. *For $k < B$, the competitive ratio is at least* $\frac{2k}{k+1}$ *for* PO *and at least* $\frac{2k-1}{k}$ *for* LPO.

Proof (Case 1. $k \geq B$.) In this case, the same hard instance works for both PO and LPO with a minor modification. Consider the following sequence of arriving packets: on step 1, there arrives a packet with B required work followed by a packet with a single required cycle; on steps $2..B-2$, $B-2$ more packets with a single required processing cycle; on step $B-1$, B packets with a single processing cycle, and then no packets until step $2B-1$, when the sequence is repeated. Under this sequence of inputs, the queues will work as follows ($\#ALG$ denotes the number of packets processed by ALG).

t	Arriving	IB_t^{PO}	# PO	IB_t^{OPT}	# OPT
1	$\boxed{1}\ \boxed{B}$	$\boxed{1}\ \boxed{B}$	0	$\boxed{1}$	1
2	$\boxed{1}$	$\boxed{1}\ \boxed{1}\ \boxed{B-1}$	0	$\boxed{1}$	2
...		...		...	
$B-2$	$\boxed{1}$	$\boxed{1}\ \ldots\ \boxed{1}\ \boxed{2}$	0	$\boxed{1}$	$B-2$
$B-1$	$\boxed{1} \times B$	$\boxed{1}\ \ldots\ \boxed{1}\ \boxed{1}$	1	$\boxed{1}\ \ldots\ \boxed{1}\ \boxed{1}$	$B-1$
...		...		...	
$2B-1$		$\emptyset$	B	$\emptyset$	$2B-2$

Thus, at the end of this sequence PO has processed B packets, while OPT has processed $2B-2$, and the sequence repeats itself, making this ratio asymptotic. For the LPO case, there is a possibility for OPT to process B more packets: at time moment $2B-1$, when $\text{IB}^{\text{LPO}} = B \times \boxed{0}$, there arrive $B \times \boxed{1}$, and LPO has to drop them.

Case 2.1 $k < B$, algorithm PO. In this case, we need to refine the previous construction; for simplicity, we assume that $k \ll B \to \infty$, and B divides everything it has to divide.

1. On step 1, there arrive $(1-\alpha)B$ packets of required work k followed by αB packets with required work 1 (α is a constant to be determined later). PO accepts all packets, while OPT rejects packets with required work k and only accepts packets with required work 1.
2. On step αB, OPT's queue becomes empty, while PO has processed $\frac{\alpha B}{k}$ packets, so it has $\frac{\alpha B}{k}$ free spaces in the queue. Thus, there arrive $\frac{\alpha B}{k}$ new packets of required work 1.
3. On step $\alpha B\left(1+\frac{1}{k}\right)$, OPT's queue is empty again, and there arrive $\frac{\alpha B}{k^2}$ new packets of required work 1.
4. ...
5. When PO is out of packets with k processing cycles, its queue is full of packets with 1 processing cycle, and OPT's queue is empty. At this point, there arrive B new packets with a single processing cycle, they are processed, and the entire sequence is repeated.

In order for this sequence to work, we need to have

$$\alpha B\left(1+\frac{1}{k}+\frac{1}{k^2}+\ldots\right)=k\,(1-\alpha)\,B.$$

Solving for α, we get $\alpha = 1-\frac{1}{k}$. During the sequence, OPT has processed $\alpha B\left(1+\frac{1}{k}+\frac{1}{k^2}+\ldots\right)+B = 2B$ packets, while PO has processed $(1-\alpha)$ $B+B=\left(1+\frac{1}{k}\right)B$ packets, so the competitive ratio is $\frac{2}{1+\frac{1}{k}}$. Note that the two competitive ratios, $\frac{2}{1+\frac{1}{k}}$ and $2\left(1-\frac{1}{B}\right)$, match when $k=B-1$.

Case 2.2 $k<B$, algorithm LPO. In this case, we can use an example similar to the previous one, but simpler since there is no extra profit to be had from an iterative construction.

1. On step 1, there arrive $(1-\alpha)B$ packets with k processing cycles followed by αB packets with a single processing cycle (α is a constant to be determined later). LPO accepts all packets, while OPT rejects packets with required work of k and only accepts packets with a single processing cycle.
2. On step αB, OPT's queue becomes empty, and there arrive βB new packets of required work 1.
3. On step $(\alpha+\beta)\,B$, OPT's queue is empty again, and LPO's queue consists of B packets with required work 0. At this point, there arrive B new packets with required work 1, they are processed, and the entire sequence is repeated.

In order for this sequence to work, we need to have

$$\left(\beta+\frac{\alpha+\beta}{k}\right)B=(1-\alpha)\,B,$$

OPT has processed $(\alpha+\beta)B$ extra packets, and from this equation we get

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

During the sequence, OPT has processed $B\left(1+\left(1+\frac{1}{k-1}\right)^{-1}\right)$ packets, and LPO has processed B packets, yielding the necessary bound. □

For large k (of the order $k\approx B^n$, $n>1$), logarithmic lower bounds follow.

Theorem 4 *The competitive ratio of* PO *(*LPO*) is at least* $\lfloor\log_B k\rfloor+1-O\left(\frac{1}{B}\right)$.

Proof We proceed by induction on B. For the induction base, we begin with the basic construction that works for $k=\Omega\left(B^2\right)$.

Lemma 1 *For $k \geq (B-1)(B-2)$, the competitive ratio of* PO *is at least $\frac{3B}{B+1}$; for* LPO*, the competitive ratio is at least 3.* □

Proof This time, we begin with the following buffer state:

$$\boxed{1}\boxed{2}\boxed{3}\boxed{4}\ \ldots\ \boxed{B-1}\boxed{B(B-1)/2}.$$

Over the next $B(B-1)/2$ steps, PO (LPO) keeps processing the first packet, while OPT, dropping the first packet, processes all the rest (their sizes sum up to the size of the first one). Thus, after $B(B-1)/2$ steps OPT's queue is empty, and PO's (LPO's) queue looks like

$$\boxed{\ }\boxed{1}\boxed{2}\boxed{3}\ \ldots\ \boxed{B-1}.$$

Over the next B steps, B packets of size 1 arrive in the system. On each step, PO (LPO) drops the packet from the head of the queue since it is the largest one, while OPT keeps processing packets as they arrive.

Thus, at the end of $B(B-1)/2+B$ steps, PO (LPO) has a queue full of $\boxed{1}$'s and OPT has an empty queue; moreover, PO (LPO) has processed only one packet (zero packets), while OPT has processed $2B$ packets. Now we have B packets of size 1 arriving, and after that they are processed and the sequence is repeated, so PO (LPO) processes $B+1$ packets (B packets) and OPT processes $3B$ packets per iteration. □

If k grows further, we can iterate upon this construction to get better bounds. For the induction step, suppose that we have already proven a lower bound of $n - O\left(\frac{1}{B}\right)$, and the construction requires maximal required work per packet less than $S = \Omega\left(B^{n-1}\right)$.

Let us now use the construction from Lemma 1, but add S to every packet's required work and, consequently, $S(B-1)$ to the first packet's required work:

$$\boxed{1+S}\boxed{2+S}\boxed{3+S}\boxed{4+S}\ \ldots\ \boxed{B-1+S}\boxed{(B+2S)(B-1)/2}.$$

For the first $(B+2S)(B-1)/2$ steps, this works exactly like the previous construction: OPT processes all packets except the first while PO (LPO) is processing the first packet. After that, OPT's queue is empty, and PO's (LPO's) queue is

$$\boxed{\ }\boxed{1+S}\boxed{2+S}\boxed{3+S}\ \ldots\ \boxed{B-1+S}.$$

Now we can add packets from the previous construction (all at once), and OPT will just take them into its queue, while PO (LPO) will replace all existing packets from its queue with new ones. Thus, the situation has been reduced to the beginning of the previous construction, but this time, PO (LPO) has already processed one packet and OPT has already processed $B-1$ packets.

This completes the proof of Theorem 4.

3.2 Upper Bound on the Competitive Ratio of LPO

We already know that the performance of LPO and PO is incomparable in the worst case (see Theorem 1), and it remains an interesting open problem to show an upper bound on the competitive ratio of PO. In this section we provide the first known upper bound of LPO. Specifically, we prove the following theorem.

Theorem 5 LPO *is at most* $(\max\{1, \ln k\} + 2 + o(1))$*-competitive.*

Recall that LPO does not transmit any packet until all packets in the buffer have zero processing cycles left. The definition of LPO allows for a well-defined accounting infrastructure. In particular, it helps us define an *iteration* during which we will count the number of packets transmitted by the optimal algorithm and compare it to the contents of LPO's buffer. The first iteration begins with the first arrival. An iteration ends when all packets in the LPO buffer have zero processing passes left. Each subsequent iteration begins after all LPO packets from the previous iteration have been transmitted. We call an iteration *congested* if LPO's buffer has been full during this iteration.

First of all, note that during an uncongested iteration no algorithm can gain over LPO because LPO processes all arrived packets (it does not push anything out). Therefore, in what follows we assume that an iteration is always congested (uncongested iterations can only improve the resulting upper bound).

We assume that OPT never pushes out packets and it is work-conserving; without loss of generality, every optimal algorithm can be assumed to have these properties since it has access to the input sequence *a priori*. Further, we enhance OPT with an additional property: at the end of each iteration, OPT flushes out all packets remaining in its buffer from the previous iteration (for free, with extra gain to its throughput). Clearly, the enhanced version of OPT is no worse than the optimal algorithm since both properties provide additional advantages to OPT versus the original optimal algorithm. In what follows, we will compare LPO with this enhanced version of OPT for the purposes of an upper bound.

To avoid ambiguity for the 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 every packet independently, in the sequence in which they arrive (although they might share the same actual time slot).

In what follows, we consider a congested iteration I that begins at time t_{beg} and ends at time t_{end} (see Fig. 3). We denote by t_{con} the first time moment when LPO's buffer is congested.

Lemma 2 *The following statements hold:*

(1) *during* I*, the buffer occupancy of* LPO *is at least the buffer occupancy of* OPT*;*
(2) *if during a time interval* $[t, t']$*,* $t_{\text{beg}} \le t \le t' \le t_{\text{con}}$*, there is no congestion in LPO's buffer then during* $[t, t']$ *OPT transmits at most* $|\text{IB}_{t'}^{\text{LPO}}|$ *packets and LPO does not transmit any packets.*

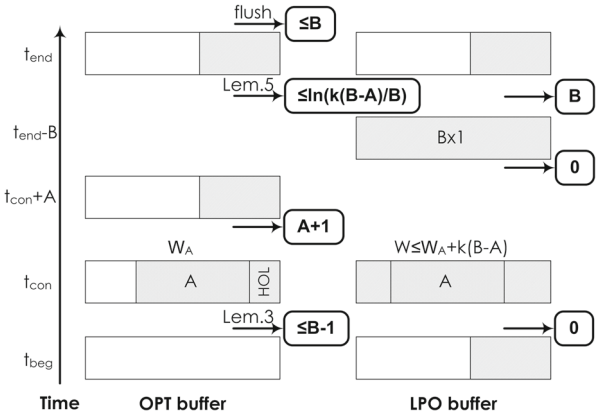

Fig. 3 The anatomy of a congested iteration. Time flows bottom up. At time t_{beg}, OPT buffer is empty. At t_{con}, the first congestion in LPO buffer occurs, and OPT has A packets plus a possible HOL packet. At time $t_{\text{end}} - B$, LPO starts transmitting. At time t_{end} (end of iteration), OPT flushes out whatever it has in the buffer for free, and new iteration begins

Proof

(1) LPO takes as many packets as it can until its buffer is full and once full it remains so for the rest of the iteration. Therefore, its buffer is at least as full as OPT's during an iteration.
(2) Since during $[t, t']$ there is no congestion and since LPO is greedy, LPO buffer contains all packets that arrive after t, and OPT cannot transmit more packets than have arrived. LPO cannot transmit during this time since it transmits only at the end of an iteration, and the iteration is assumed to be congested.

□

Let us now go over the iteration (bottom up on Fig. 3) and count the packets that OPT can process on each step.

Lemma 3

(1) *During* $[t_{\text{beg}}, t_{\text{con}}]$, OPT *processes at most* $B - 1$ *packets.*
(2) *For every packet* p *in* OPT*'s buffer at time* t_{con} *except perhaps the HOL packet, there is a corresponding packet* q *in* LPO*'s buffer with* $r(q) \leq r(p)$.

Proof During $[t_{\text{beg}}, t_{\text{con}}]$, there arrive exactly B packets (because LPO does not transmit any packets and becomes congested at t_{con}). Moreover, OPT cannot process all B packets because then LPO would also have time to process them, and the iteration would be uncongested. Item (2) is equally obvious: every packet in OPT's buffer also resides in LPO's buffer because LPO has not dropped anything yet at time t_{con}; $r(q) \leq r(p)$ because LPO may have processed some packets partially. □

We denote by A the number of non-HOL packets in OPT's buffer at time t_{con}; by W_A, their total required processing. We denote by M_t the maximal number of residual processing cycles among all packets in LPO's buffer at time t that belong to the current iteration;[3] by W_t, the total residual work for all packets in LPO's buffer at time t. Lemma 3 shows that LPO's buffer at time t_{con} contains A corresponding packets, so

$$W_{t_{\text{con}}} \leq W_A + (B - A)k.$$

Moreover, over the next W_A time slots OPT will be processing these A packets and LPO, being congested, will also not be idle, so at time $t_{\text{con}} + W_A$ we will have $W_{t_{\text{con}}+A} \leq (B - A)k$ (we give OPT its HOL packet for free, so OPT processes $A + 1$ packets over $[t_{\text{con}}, t_{\text{con}} + A]$).

Lemma 4 *For every packet accepted by* OPT *at time* $t \in [t_{\text{con}}, t_{\text{end}}]$ *and processed by* OPT *during time interval* $[t', t'']$, $t_{\text{con}} \leq t' \leq t'' \leq t_{\text{end}}$, $W_{t''} \leq W_{t-1} - M_t$.

Proof If LPO's buffer is full then a packet p accepted by OPT either pushes out a packet in LPO's buffer or is rejected by LPO. If p pushes a packet out, then the total work W_{t-1} is immediately reduced by $M_t - r_t(p)$. Moreover, after processing p, $W_{t''} \leq W_{t-1} - (M_t - r_t(p)) - r_t(p) = W_{t-1} - M_t$. If, on the other hand, p is rejected by LPO then $r_t(p) \geq M_t$, and thus $W_{t''} \leq W_{t-1} - r_t(p) \leq W_{t-1} - M_t$. □

Next we denote by $f(B, W)$ the maximal number of packets that OPT can transmit during $[t, t']$, $t_{\text{con}} \leq t \leq t' \leq t_{\text{end}}$, where $W = W_{t-1}$. The next lemma is crucial for the proof; it shows that OPT cannot have more than logarithmic (in k) gain over LPO during the congestion period. Note that the statement of Lemma 5 could have been simplified to $f(B, W) \leq B \ln \frac{W}{B}$, which would suffice to prove Theorem 5. However, we have decided to spell out the strongest result available to us. In particular, this slightly stronger formulation may serve to close the gap between lower and upper bounds either in this model or in subsequent modifications of it (since Lemma 5 is quite general, and its ideas can be applied to other lazy policies).

Lemma 5 *For every* $\epsilon > 0$, $f(B, W) \leq \frac{B-1}{1-\epsilon} \ln \frac{W}{B}$ *for* B *sufficiently large.*

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 $t_{\text{end}} - B$. At any time t, $M_t \geq \frac{W_t}{B}$: the maximal required processing is no less than the average. By Lemma 4, 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 $W\left(1 - \frac{1}{B}\right)$.

[3]This qualification deals with a boundary case: during the last B time slots of an iteration, when LPO is transmitting, we set $M_t = 1$ for $t \in [t_{\text{end}} - B, t_{\text{end}}]$ 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.

We now prove the statement by induction on W. The base is trivial for $W = B$ since all packets are already 1's.

The induction hypothesis is now that after one packet is processed by OPT, there cannot be more than $f\left(B, \frac{W}{B}\left(1-\frac{1}{B}\right)\right) \leq \frac{B-1}{1-\epsilon}\ln\left[\frac{W}{B}\left(1-\frac{1}{B}\right)\right]$ packets left, and for the induction step we have to prove that

$$\frac{B-1}{1-\epsilon}\ln\left[\frac{W}{B}\left(1-\frac{1}{B}\right)\right]+1 \leq \frac{B-1}{1-\epsilon}\ln\frac{W}{B}.$$

This is equivalent to

$$\ln\frac{W}{B} \geq \ln\left[\frac{W}{B}\frac{B-1}{B}e^{\frac{1-\epsilon}{B-1}}\right],$$

and this holds asymptotically because for every $\epsilon > 0$, we have $e^{\frac{1-\epsilon}{B-1}} \leq \frac{B}{B-1}$ for B sufficiently large. □

Applying Lemma 5 to the time $t_{\text{con}} + A$, we get the following.

Corollary 1 *For every $\epsilon > 0$, the total number of packets processed by* OPT *between t_{con} and t_{end} in a congested iteration does not exceed*

$$A + 1 + (B + o(B))\ln\frac{(B-A)k}{B}.$$

Now we are ready to prove Theorem 5.

Proof Consider an iteration I that begins at time t_{beg} and ends at time t_{end}. If I is uncongested then, as we have already noted, OPT cannot transmit more than $\left|\text{IB}_t^{\text{LPO}}\right|$ packets during I.

Consider an iteration I first congested at time t_{con}, $t_{\text{beg}} \leq t_{\text{con}} \leq t_{\text{end}}$. By Lemma 2(2), during $[t_{\text{beg}}, t_{\text{con}})$OPT can transmit at most $B-1$ packets, leaving $A+1$ packets in its buffer. By Corollary 1, OPT processes at most $A+1+\frac{B-1}{1-\epsilon}\ln\frac{(B-A)k}{B} + o\left(B\ln\frac{(B-A)k}{B}\right)$ packets during $[t_{\text{con}}, t_{\text{end}}]$, and then 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

$$2B + A + (B + o(B))\ln\frac{(B-A)k}{B}.$$

It is now easy to check that for every $1 \leq A \leq B-1$ the competitive ratio is bounded by the the theorem's statement. □

3.3 The Case of Two Required Processing Values

In this section, we consider a special case when there are only two possible required processing values, 1 and k, so there are only two kinds of packets, $\boxed{1}$ and $\boxed{k}$. This case often occurs in practice; for instance, $\boxed{1}$ may represent "commodity" packets that are processed to completion while $\boxed{k}$ corresponds to packets requiring

advanced processing of some kind. In this special case, PO and LPO have even better competitiveness guarantees, as the bounds below indicate.

In what follows, we show a lower bound of $2 - \frac{1}{k}$ and a matching upper bound of $2 + \frac{1}{B}$ on the competitive ratio of LPO.

Theorem 6 (1) *The competitive ratio of* LPO *in the two-valued case is at least* $2 - \frac{1}{k}$ *for* $B \geq k$.

(2) *The competitive ratio of* PO *in the two-valued case is at least* $\frac{2}{1+\frac{1}{k}}$ *for* $B \geq k$.

Proof

1. Similar to Theorem 3, the first burst for LPO consists of $\frac{B}{k} \times \boxed{k}$ followed by $\left(B - \frac{B}{k}\right)\boxed{1}$. Over the next $\left(B - \frac{B}{k}\right)$ steps, OPT processes all $\boxed{1}$'s while LPO leaves only $\boxed{1}$'s and $\boxed{0}$'s in the queue. Then, $B \times \boxed{1}$ arrive, and both algorithms process B more packets, getting the ratio of $2 - \frac{1}{k}$.
2. For PO, the first burst is the same but we wait for B steps; after B steps, PO has processed $\frac{B}{k}$ packets while OPT has processed B packets, and we add $B \times \boxed{1}$, getting the bound.

□

Theorem 7 *In the two-valued case,* LPO *is at most* $\left(2 + \frac{1}{B}\right)$*-competitive.*

Proof Following the notation from the proof of Theorem 5, consider a congested iteration starting at time t_{beg}, getting its first congestion at time t_{con}, and ending at time t_{end}. The proof of the upper bound is very similar to the proof of Theorem 5, with two important differences. Lemma 2 goes through without change.

The first difference is that Lemma 4 now becomes stronger.

Lemma 6 *For every packet accepted by* OPT *at time* $t \in [t_{\text{con}}, t_{\text{end}}]$ *and processed by* OPT *during time interval* $[t', t'']$, $t_{\text{con}} \leq t' \leq t'' \leq t_{\text{end}}$, $W_{t''} \leq W_{t-1} - k$ *in all cases except possibly the very last* $^{non}\boxed{1}$ *packet in* LPO*'s buffer.* □

Proof As long as LPO buffer contains at least one $\boxed{k}$, $M_t = k$. □

Second, the two-valued case allows for more accurate analysis of the congested period. Suppose that during $[t_{\text{beg}}, t_{\text{con}}]$ (before first congestion) OPT has processed $\alpha \times \boxed{1}$ *extra* packets (note that there cannot be any extra $\boxed{k}$ packets for OPT). This means that at time t_{con}, LPO's buffer contains at least $\alpha \times \boxed{1}$ (LPO never pushes out a $\boxed{1}$ if it is possible to process it), and LPO has already spent at least α time slots on its $\boxed{k}$'s. Therefore, $W_{t_{\text{con}}} \leq \alpha + (B - \alpha)k - \alpha = (B - \alpha)k$. Thus, during $[t_{\text{con}}, t_{\text{end}}]$ there can be at most $B - \alpha + 1$ steps of Lemma 6, and as a result, OPT processes

at most $B - \alpha + 1$ packets during $[t_{\text{con}}, t_{\text{end}}]$ and then gets the final flush of B more packets. Therefore, the resulting ratio is at most

$$\frac{\alpha + B - \alpha + 1 + B}{B} = 2 + \frac{1}{B}.$$

4 Simulation Study

In this section, we consider the proposed policies (both push-out and non-push-out) for FIFO buffers and conduct a simulation study in order to further explore and validate their performance. Namely, we compare the performance of NPO, PO, and LPO in different settings. It was shown in [15] that a push-out algorithm that processes packets with less required processing first is optimal, so in what follows we denote it by OPT*. Clearly, OPT in the FIFO queueing model does not outperform OPT*.

Publicly available traffic traces (such as CAIDA [6]) do not contain, to the best of our knowledge, information on the processing requirements of packets. Furthermore, these requirements are difficult to extract since they depend on the specific hardware and NP configuration of the network elements. Another handicap of such traces is that they provide no information about time-scale, and specifically, 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, as well as traffic burstiness. We therefore perform our simulations on synthetic traces.

Our traffic is generated from 100 independent sources, each generated by an on-off Markov modulated Poisson process (MMPP) which we use to simulate bursty traffic. The choice of parameters is governed by average arrival load, which is determined by the product of the average packet arrival rate and the average number of processing cycles required by packets. In our simulations, each source has average packet rate of $\frac{1}{2\text{l}}\lambda_{\text{on}}$, where λ_{on} is the parameter governing traffic generation while in the on state. Each source also has a fixed required processing value for every emitted packet; these values are distributed evenly across $\{1, \ldots, k\}$ (k being the maximum amount of processing required by any packet). We conducted our simulations for 500,000 time slots, and allowed various parameters to vary in each set of simulations, in order 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 $[1, 40]$ and the underlying source intensity λ_{on} in the range $[0.005, 0.25]$, we evaluate the performance of our algorithms in settings ranging from underload to extreme overload, validating their performance in various traffic scenarios. Figure 4 shows simulation results. The vertical axis always represents the ratio between the algorithm's performance and OPT* performance 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, four sets of simulations in total.

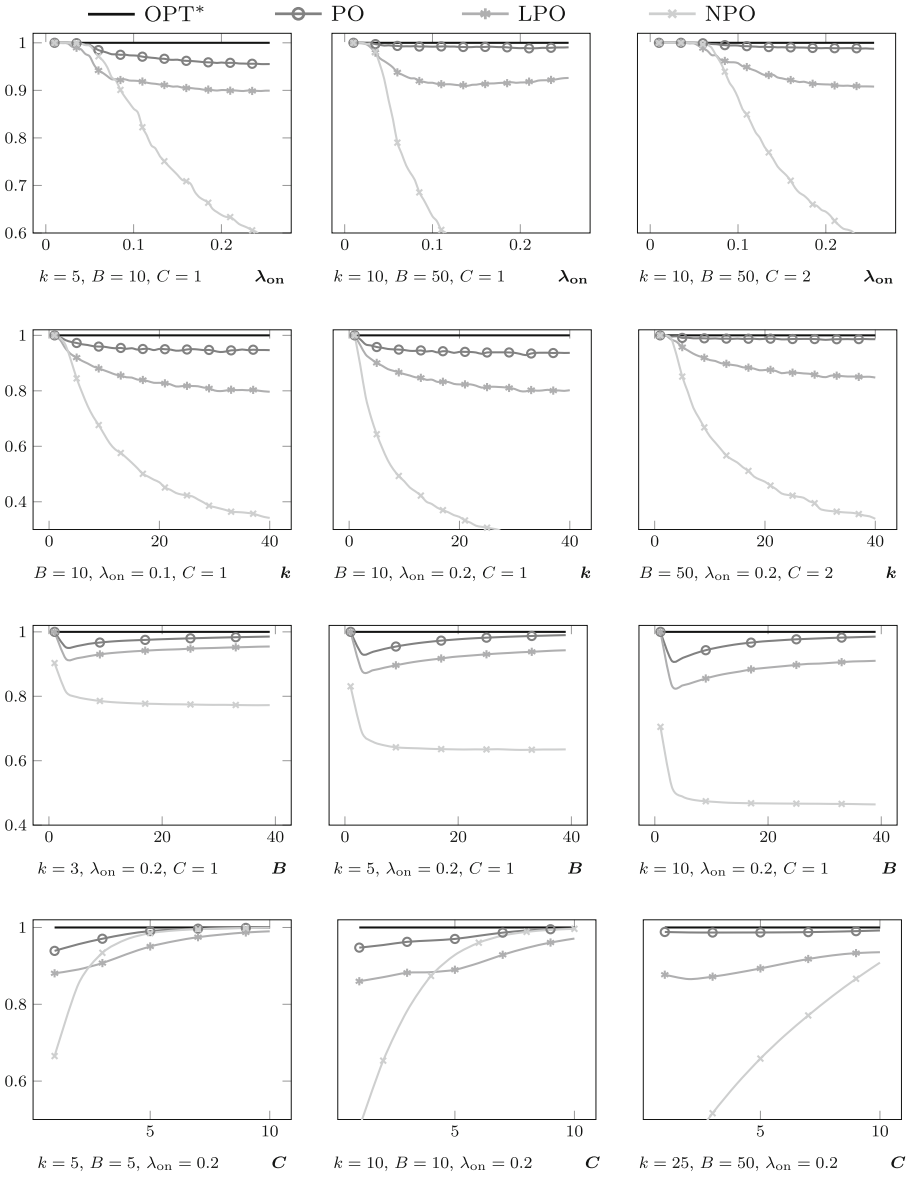

Fig. 4 Performance ratio of online algorithms versus optimal as a function of parameters: row 1, of λ_{on}; row 2, of k; row 3, of B; row 4, of C. The y-axis on all graphs shows the competitiveness vs. OPT*

Variable Intensity and Variable Maximum Number of Required Processing Cycles Both the first and second set of simulations amount to testing different policies under gradually increasing processing requirements. The first and second row of graphs on Fig. 4 show that OPT* keeps outperforming LPO and NPO more and more as k grows; however, the difference in processing order between OPT* and PO does

Table 1 Results summary: lower and upper bounds

Algorithm/family	Case	Lower bound	Upper bound
NPO		k	k
PO	$k < B$	$\frac{2k}{k+1}$	open problem
	$k \geq B$	$2\left(1-\frac{1}{B}\right)$	
	$k \gg B$	$\lfloor \log_B k \rfloor + 1 - O\left(\frac{1}{B}\right)$	
	2-valued, $k \leq B$	$\frac{2k}{k+1}$	
LPO	$k < B$	$2-\frac{1}{k}$	$(\max\{1, \ln k\} + 2 + o(1))$
	$k \geq B$	$3\left(1-\frac{1}{B}\right)$	$(\max\{1, \ln k\} + 2 + o(1))$
	$k \gg B$	$\lfloor \log_B k \rfloor + 1 - O\left(\frac{1}{B}\right)$	$(\max\{1, \ln k\} + 2 + o(1))$
	2-valued, $k \leq B$	$2-\frac{1}{k}$	$\left(2+\frac{1}{B}\right)$

not matter much. NPO results show that non-push-out policies cope very badly with overload scenarios, as expected.

Variable Buffer Size In this set of simulations we evaluated the performance of our algorithms for variable values of B in the range $[1, 40]$. Throughout our simulations we again assumed a single core ($C = 1$) and evaluated different values of k. The third row on Fig. 4 presents our results. Unsurprisingly, the performance of all algorithms significantly improves as the buffer size increases; the difference between OPT* and two other push-out algorithms visibly reduces, but, of course, it would take a huge buffer for NPO to catch up (one would need to virtually remove the possibility of congestion).

Variable Number of Cores In this set of simulations we evaluated the performance of our algorithms for variable values of C in the range $[1, 10]$. The bottom row of Fig. 4 presents our results; the performance of all algorithms, naturally, improves drastically as the number of cores increases. There is an interesting phenomenon here: push-out capability becomes less important since buffers are congested less often, but LPO keeps paying for its "laziness"; so as C grows, eventually NPO outperforms LPO. The increase in the number of cores essentially provides the network processor (NP) with a speedup proportional to the number of cores (assuming the average arrival rate remains constant).

5 Conclusion

In this paper, we provide performance guarantees for NP buffer scheduling algorithms with FIFO queueing for packets with heterogeneous required processing. The objective is to maximize the number of transmitted packets under various settings such as push-out and non-push-out buffers. We validate our results by simulations. Table 1 summarizes our contributions. As future work, it will be interesting to show

an upper bound for the PO algorithm and try to close the gaps between lower and upper bounds of the proposed on-line algorithms.

Acknowledgments The work of S.I. Nikolenko was supported by the Basic Research Program of the National Research University Higher School of Economics, 2015, grant no. 78. We also thank the anonymous referees for many useful comments that have allowed us to improve the paper.

References

1. Aiello, W., Mansour, Y., Rajagopolan, S., Rosén, A.: Competitive queue policies for differentiated services. J. Algorithms **55**(2), 113–141 (2005)
2. Azar, Y., Litichevskey, A.: Maximizing throughput in multi-queue switches. Algorithmica **45**(1), 69–90 (2006)
3. Azar, Y., Richter, Y.: An improved algorithm for CIOQ switches. ACM Trans. Algorithms **2**(2), 282–295 (2006)
4. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press (1998)
5. Brucker, P., Heitmann, S., Hurink, J., Nieberg, T.: Job-shop scheduling with limited capacity buffers. OR Spectrum **28**(2), 151–176 (2006)
6. CAIDA – the cooperative association for internet data analysis. [Online] http://www.caida.org/
7. Cavium: OCTEON II CN68XX multi-core MIPS64 processors, product brief (2010). [Online] http://www.caviumnetworks.com/OCTEON-II_CN68XX.html
8. Chuprikov, P., Nikolenko, S.I., Kogan, K.: Priority Queueing with Multiple Packet Characteristics. In: IEEE INFOCOM. to appear, pp. 1–9 (2015)
9. Cisco: The Cisco QuantumFlow processor, product brief (2010). [Online] http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_overview_c22-448936.html
10. Englert, M., Westermann, M.: Lower and upper bounds on FIFO buffer management in QoS switches. Algorithmica **53**(4), 523–548 (2009)
11. Eugster, P., Kogan, K., Nikolenko, S.I., Sirotkin, A.V.: Shared Memory Buffer Management for Heterogeneous Packet Processing. In: IEEE 34th International Conference on Distributed Computing Systems (ICDCS 2014), pp. 471–480 (2014)
12. EZChip: NP-4 network processor, product brief (2010). [Online] http://www.ezchip.com/p_np4.htm
13. Goldwasser, M.: A survey of buffer management policies for packet switches. SIGACT News **41**(1), 100–128 (2010)
14. Kesselman, A., Kogan, K.: Nonpreemptive Scheduling of Optical Switches. IEEE Trans. Commun. **55**(6), 1212–1219 (2007)
15. Keslassy, I., Kogan, K., Scalosub, G., Segal, M.: Providing performance guarantees in multipass network processors. IEEE/ACM Trans. Networking **20**(6), 1895–1909 (2012)
16. Kesselman, A., Kogan, K., Segal, M.: Improved competitive performance bounds for CIOQ switches. Algorithmica **63**(1–2), 411–424 (2012)
17. Kesselman, A., Kogan, K., Segal, M.: Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing. Distrib. Comput. **23**(3), 163–175 (2010)
18. Kesselman, A., Kogan, K., Segal, M.: Best Effort and Priority Queuing Policies for Buffered Crossbar Switches (2012)
19. Kesselman, A., Lotker, Z., Mansour, Y., Patt-Shamir, B., Schieber, B., Sviridenko, M.: Buffer overflow management in QoS switches. SIAM J. Comput. **33**(3), 563–583 (2004)
20. Kogan, K., López-Ortiz, A., Nikolenko, S.I., Sirotkin, A.V.: A taxonomy of semi-FIFO policies. In: Proceedings of the 31st IEEE International Performance Computing and Communications Conference (IPCCC 2012), pp. 295–304 (2012)
21. Kogan, K., López-Ortiz, A., Nikolenko, S.I., Sirotkin, A.V., Tugaryov, D.: FIFO queueing policies for packets with heterogeneous processing. In: Proceedings of the 1st Mediterranean Conference on Algorithms (MedAlg 2012), Lecture Notes in Computer Science. arXiv:1204.5443 [cs.NI], vol. 7659, pp. 248–260 (2012)

22. Kogan, K., López-Ortiz, A., Nikolenko, S.I., Sirotkin, A.V.: Multi-queued network processors for packets with heterogeneous processing requirements. In: Proceedings of the Fifth International Conference on Communication Systems and Networks, (COMSNETS), pp. 1–10 (2013)
23. Kogan, K., López-Ortiz, A., Nikolenko, S.I., Scalosub, G., Segal, M.: Balancing work and size with bounded buffers. In: Proceedings of the Sixth International Conference on Communication Systems and Networks, (COMSNETS). [Online] arXiv:1202.5755, pp. 1–8 (2014)
24. Kogan, K., Nikolenko, S.I., Keshav, S., López-Ortiz, A.: Efficient demand assignment in multi-connected microgrids with a shared central grid. In: SustainIT, pp. 1–5 (2013)
25. Leonardi, S., Raz, D.: Approximating total flow time on parallel machines. In: STOC, pp. 110–119 (1997)
26. Mansour, Y., Patt-Shamir, B., Lapid, O.: Optimal smoothing schedules for real-time streams. Distrib. Comput. **17**(1), 77–89 (2004)
27. Motwani, R., Phillips, S., Torng, E.: Non-clairvoyant scheduling. Theor. Comput. Sci. **130**(1), 17–47 (1994)
28. Muthukrishnan, S.M., Rajaraman, R., Shaheen, A., Gehrke, J.E.: Online scheduling to minimize average stretch. SIAM J. Comput. **34**(2), 433–452 (2005)
29. Nikolenko, S.I., Kogan, K.: Single and multiple buffer processing. Encyclopedia of Algorithms, 1–9 (2015)
30. Pruhs, K.: Competitive online scheduling for server systems. SIGMETRICS Perform. Eval. Rev. **34**(4), 52–58 (2007)
31. Ruiz, R., Vázquez-Rodrígue, J.A.: The hybrid flow shop scheduling problem. Eur. J. Oper. Res. **205**(1), 1–18 (2010)
32. Schrage, L.: A proof of the optimality of the shortest remaining processing time discipline. Oper. Res. **16**, 687–690 (1968)
33. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM **28**(2), 202–208 (1985)
34. Wolf, T., Pappu, P., Franklin, M.A.: Predictive scheduling of network processors. Comput. Netw. **41**(5), 601–621 (2003)