FIFO Queueing Policies for Packets with Heterogeneous Processing

Kirill Kogan[1], Alejandro López-Ortiz[1], Sergey I. Nikolenko[2,3⋆], Alexander V. Sirotkin[4,3⋆] and Denis Tugaryov[3⋆]

[1] School of Computer Science, University of Waterloo
[2] Steklov Mathematical Institute, nab. r. Fontanka, 27, St. Petersburg, Russia
[3] St. Petersburg Academic University, ul. Khlopina, 8, korp. 3, St. Petersburg, Russia,
[4] St. Petersburg Institute for Informatics and Automation of the RAS, 14 Line VO, 39, St. Petersburg, Russia

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 total number of successfully transmitted packets. We consider both push-out (when the policy is permitted to drop already admitted packets) and non-push-out cases. In particular, we provide analytical guarantees for the throughput performance of our algorithms. We further conduct a comprehensive simulation study which experimentally validates the predicted theoretical behaviour.

Keywords: Scheduling, Buffer Management, First-In-First-Out Queueing, Switches, Online Algorithms, Competitive Analysis.

1 Introduction

This work is mostly motivated by buffer management problems within Network Processors (NPs) in a packet-switched network. Such NPs are responsible for complex packet processing tasks in modern high-speed routers, including, to name just a few, forwarding, classification, protocol conversion, and intrusion detection. Common NPs usually rely on multi-core architectures, where multiple cores perform various processing tasks required by the arriving traffic. Such architectures may be based on a pipeline of cores [1], a pool of identical cores [2–4], or a hybrid pool pipeline [5]. In response to operator demands, packet processing needs are becoming more heterogeneous, as NPs need to cope with more complex tasks such as advanced VPN services and hierarchical classification for QoS, among others. Unlike general purpose processors, modern NPs employ run-to-completion processing. Recent results in data path provisioning provide a possibility to have information about future required processing *a priori* (for instance, this is possible in one of the modes of the OpenFlow protocol [6]). In this work, we consider a model that captures the characteristics of this architecture. We evaluate the performance of such systems for the case when information about required processing is available *a priori*. The main concern in this setting is to maximize the throughput attainable by the NP, measured by the total number of packets successfully processed by the system.

In what follows, we adopt the terminology used to describe buffer management problems. We focus our attention on a general model where we are required to manage admission control and scheduling modules of a single bounded size queue that process packets in First-In-First-Out order. In this model, arriving traffic consists of unit-sized *packets*, and each packet has a *processing requirement* (in processor cycles). A packet is successfully *transmitted* once the scheduling module 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), then the packet is lost without gain to the algorithm's throughput.

1.1 Our Contributions

In this paper, we consider the problem of managing a FIFO queue buffer of size B, where each incoming unit-sized packet requires at most k rounds of processing before it can be transmitted out. Our objective is to maximize the total number of successfully transmitted packets. 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 of processing cycles required by a packet. However, none of our algorithms needs to know the maximum number of processing cycles in advance. We discuss the non-push-out case in Section 2 and show that the on-line greedy algorithm NPO is k-competitive, and that this bound is tight. For the push-out case, we consider two algorithms: a simple greedy algorithm PO that in the case of congestion pushes out the first packet with maximal required processing, and Lazy-Push-Out (LPO) algorithm that mimics PO but does not transmit packets if there is still at least one admitted packet with more than one required processing cycle. Intuitively, it seems that PO should outperform LPO since PO tends to empty its buffer faster but we demonstrate that these algorithms are not comparable in the worst case. Although we provide a lower bound of PO, the main result of this paper deals with the competitiveness of LPO. In particular, we demonstrate that LPO is at most $\left(\ln k + 3 + \frac{o(B)}{B} \right)$-competitive. In addition, we demonstrate several lower bounds on the competitiveness of both PO and LPO for different values of B and k. These results are presented in Section 3. The competitiveness result of LPO is interesting in itself but since "lazy" algorithms provide a well-defined accounting infrastructure we hope that a similar approach can be applied to other systems in similar settings. From an implementation point of view we can define a new on-line algorithm that will emulate the behaviour of LPO and will not delay the transmission of processed packets. In Section 4 we conduct a comprehensive simulation study to experimentally verify the performance of the proposed algorithms. All proofs not appearing in the body of the paper can be found in the Appendix.

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 the arriving traffic. 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 for such settings. Although they considered a different architecture (FIFO with recycles) than the one we consider in this paper, they provided only a lower bound for the push-out FIFO case, and it remains unknown if it can be attained.

Kogan et al. [8] considered priority-based buffer management and scheduling in both push-out and non-push-out settings for heterogeneous packet sizes. Specifically, they consider two priority queueing schemes: (i) Shortest Remaining Processing Time first (SRPT) and (ii) Longest Packet first (LP). They present competitive buffer management algorithms for these schemes and provide lower bounds on the performance of algorithms for such priority queues.

The work of Keslassy et al. [7] and Kogan et al. [8], as well as our current 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 [9] which provides an excellent overview of this field). This line of research, initiated in [10, 11], 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 [10–13] and models where packets have dependencies [14, 15]. A related field that has received much attention in recent years focuses on various switch architectures and aims at designing competitive algorithms for such multi-queue scenarios; see, e.g., [16–20]. Some other works also provide experimental studies of these algorithms and further validate their performance [21].

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 [22]. Additional objectives, models, and algorithms have been studied extensively in this context [23–25]. For a comprehensive overview of competitive online scheduling for server systems, see a survey by Pruhs [26]. 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 [27], most notably to hybrid flow-shop scheduling [28] in scenarios where machines have bounded buffers but are not allowed to drop and push out tasks.

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 [29]. Although the value of k will play a fundamental role in our analysis, we note that our algorithms need not know k in advance. In what follows, we adopt the terminology used in [8]. 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 scheduler 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. However, for simplicity, in the remainder of this paper we assume the system selects a single packet for processing at any given time (i.e., $C = 1$). This simple setting suffices to show both the difficulties of the model and our algorithmic scheme. We assume discrete slotted time, where 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;
 (ii) *assignment and processing*: a single packet is selected for processing by the scheduling module;
(iii) *transmission*: packets with zero required processing left are transmitted and leave 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 overall number of packets transmitted from the queue.

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 it has admitted packets that require processing in the queue.
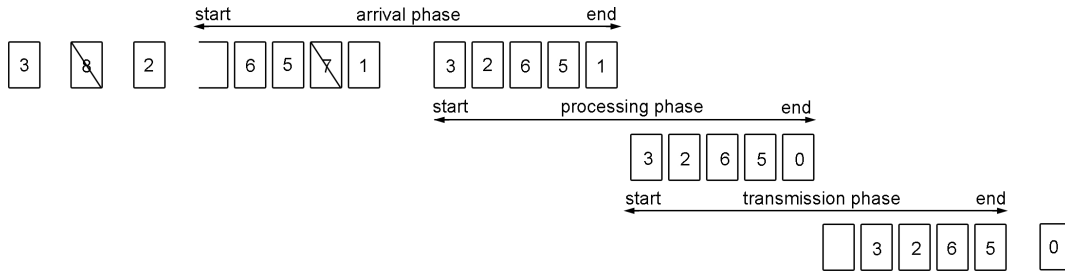
Fig. 1. Zoom in on a single time slot for greedy, push-out, and work-conserving algorithm.

We say that an arriving packet p *pushes out* a packet q that has already been accepted into the buffer iff 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 greedy and push-out case).

For an algorithm ALG and a time slot t, we denote the set of packets stored in ALG's buffer at time t by IB_t^{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, 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.

Our goal is to provide performance guarantees for various buffer management algorithms. We use competitive analysis [30, 31] when evaluating 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 successfully transmitted by an optimal solution (denoted OPT) obtained by an offline clairvoyant algorithm.

1.4 Proposed Algorithms

Algorithm 1 NPO(p): Buffer Management Policy

1: **if** buffer occupancy is less than B **then**
2:      accept p
3: **else**
4:      drop p
5: **end if**

Next we define 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. 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).

Next we introduce two push-out algorithms. The Push-Out Algorithm (PO) is also greedy and work-conserving, but now, if an arriving packet p requires less processing cycles than at least one packet in its buffer, then PO pushes out the first packet with the maximal number of processing cycles in its buffer

4

Algorithm 2 PO(p): Buffer Management Policy

1: **if** buffer occupancy is less than B **then**
2:     accept p
3: **else**
4:     let q be the first (from HOL) packet with maximal number of residual processing
5:     **if** $r_t(p) < r_t(q)$ **then**
6:         drop q and accept p according to FIFO order
7:     **end if**
8: **end if**

---

and accepts p according to FIFO order (see Algorithm 2). The second algorithm is a new Lazy-Push-Out algorithm LPO that mimics the behaviour of PO with two important differences: (i) LPO does not transmit a Head-Of-Line packet with a single processing cycle if its buffer contains at least one packet with more than one residual processing cycle, until the buffer contains only packets with a single residual processing cycle; (ii) once all packets in LPO's buffer (say there are m packets there) have a single processing cycle remaining, LPO transmits them over the next m/C processing cycles where C is the number of processing cores; observe that during this time, 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 (similarly to PO).

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, Theorem 1 shows that LPO and PO are incomparable in the worst case.

Theorem 1. *(1) There exists a sequence of inputs on which* PO *processes* $\geq \frac{3}{2}$ *times more packets than* LPO. *(2) There exists a sequence of inputs on which* LPO *processes* $\geq \frac{5}{4}$ *times more packets than* PO.

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

- first burst of B packets with required work 2 arriving at time slot $t = 1$;
- second burst of B packets with required work 1 arriving at time slot $t = B$.

By the time the second burst arrives, LPO has processed no packets, while PO has processed $\frac{B}{2}$ packets. Then both algorithms process B packets of required work 1 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.

To prove (2), suppose for simplicity that $k > \frac{B}{2}$. 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	$\boxed{2} \times B$	$\boxed{2}\ldots\boxed{2}$	0	$\boxed{2}\ldots\boxed{2}$	0
B	$\boxed{k} \times \frac{B}{2}$	$\boxed{1}\ldots\boxed{1}$	0	$\boxed{k}\ldots\boxed{k}\boxed{2}\ldots\boxed{2}$	$B/2$
$2B-1$	none	$\boxed{1}$	$B-1$	$\boxed{k}\ldots\boxed{k}\boxed{1}$	$B-1$
$2B$	$\boxed{1} \times \frac{B}{2}$	$\boxed{1}\ldots\boxed{1}$	B	$\boxed{1}\ldots\boxed{1}\boxed{k}\ldots\boxed{k}$	B
$\frac{5B}{2}-1$	none	$\boxed{1}$	$3B/2-1$	$\boxed{1}\ldots\boxed{1}\boxed{k}\ldots\boxed{\text{k-B/2+1}}$	B
$\frac{5B}{2}$	$\boxed{1} \times B$	$\boxed{1}\ldots\boxed{1}$	$3B/2$	$\boxed{1}\ldots\boxed{1}$	B
$\frac{7B}{2}$	none	$\emptyset$	$5B/2$	$\emptyset$	$2B$

Similar to (1), we can repeat this sequence. $\qquad\square$

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; its proof is given in the Appendix.

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.*

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 and prove an upper bound for LPO.

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. Proofs of Theorems 3 and 4 are given in the Appendix.

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

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 $\lfloor \log_B k \rfloor + 1 - O(\frac{1}{B})$.*

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 $\left(\ln k + 3 + \frac{o(B)}{B}\right)$-competitive.*

We remind that LPO does not transmit any packet until all packets in the buffer have exactly one processing cycle left. The definition of LPO allows for a well-defined accounting infrastructure. In particular, LPO's definition helps us to 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 a single processing pass left. Each subsequent iteration starts after the transmission of all LPO packets from the previous iteration.

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 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 LPO'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 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).

Claim. Consider an iteration I that begins at time t' and ends at time t. The following statements hold: (1) during I, the buffer occupancy of LPO is at least the buffer occupancy of OPT; (2) between two subsequent iterations I and I', OPT transmits at most $|\mathrm{IB}_t^{\mathrm{LPO}}|$ packets; (3) if during a time interval $[t',t”]$, $t' \leq t” \leq t$, there is no congestion then during $[t',t”]$ OPT transmits at most $|\mathrm{IB}_{t”}^{\mathrm{LPO}}|$ packets.

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 hence its buffer is at least as full as OPT's during an iteration. (2) By (1), at the end of an iteration the buffer occupancy of LPO is at least the buffer occupancy of OPT; moreover, all packets in LPO buffer at the end of an iteration have a single processing cycle. (3) Since during $[t',t”]$ there is no congestion and since LPO is greedy, LPO buffer contains all packets that have arrived after t, and thus, OPT cannot transmit more packets than have arrived. $\qquad\square$

We denote by M_t the maximal number of residual processing cycles among all packets in LPO's buffer at time t; by W_t, the total residual work for all packets in LPO's buffer at time t.

Lemma 1. *For every packet accepted by* OPT *at time t and processed by* OPT *during the time interval* $[t_s,t_e]$, $t \leq t_s \leq t_e$, *if* $|\mathrm{IB}_{t-1}^{\mathrm{LPO}}| = B$ *then* $W_{t_e} \leq W_{t-1} - M_t$.

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 reduced by $M_t - r_t(p)$. Moreover, after processing p, $W_{t_e} \leq W_{t-1} - (M_t - r_t(p)) - r_t(p) = W_{t-1} - M_t$. Otherwise, if p is rejected by LPO then $r_t(p) \geq M_t$, and thus $W_{t_e} \leq W_{t-1} - r_t(p) \leq W_{t-1} - M_t$. $\qquad\square$

Let t be the time of the first congestion during an iteration I that has ended at time t'. Observe that by definition, at time t, OPT flushes out all packets that were still in its buffer at time $t - 1$. We denote by $f(B,W)$ the maximal number of packets that OPT can process during $[t,t']$, where $W = W_{t-1}$.

Lemma 2. *For every $\epsilon > 0$, $f(B,W) \leq \frac{B-1}{1-\epsilon} \ln \frac{W}{B} + o(B \ln \frac{W}{B})$.*

7

Proof. By definition, LPO does not transmit packets during an iteration. Hence, if the buffer of LPO is full, it will remain full until the end of iteration. At any time t, $M_t \geq \frac{W_t}{B}$: the maximal required processing is no less than the average. By Lemma 1, 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 $W\left(1 - \frac{1}{B}\right)$.

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

Now we are ready to prove Theorem 5.

Proof (of Theorem 5). Consider an iteration I that begins at time t' and ends at time t.

1. LPO*'s buffer is not congested during I.* In this case, by Claim 3.2(3) OPT cannot transmit more than $|\,\mathrm{IB}_t^{\mathrm{LPO}}\,|$ packets during I.
2. *During I,* LPO*'s buffer is first congested at time t'', $t' \leq t'' \leq t$.* If during I OPT transmits less than B packets then we are done. By Claim 3.2(3), during $[t',t'']$ OPT can transmit at most B packets. Moreover, at most B packets are left in OPT buffer at time $t'' - 1$. By Lemma 2, during $[t'',t]$ LPO transmits at most $(\ln k + \frac{o(B)}{B})B$ packets (because $W \leq kB$), so the total amount over a congested iteration is at most $(\ln k + 2 + \frac{o(B)}{B})B$ packets.

Therefore, during an iteration OPT transmits at most $(\ln k + 2 + o(1))|\,\mathrm{IB}_t^{\mathrm{LPO}}\,|$ packets. Moreover, by Claim 3.2(2), between two subsequent iterations OPT can transmit at most $|\,\mathrm{IB}_t^{\mathrm{LPO}}\,|$ additional packets. Thus, LPO is at most $\ln k + 3 + \frac{o(B)}{B}$-competitive. □

The bound shown in Theorem 5 is asymptotic. To cover small values of B, we show a weaker bound ($\log_2 k$ instead of $\ln k$) on inputs where LPO never pushes out packets that are currently being processed.

The following theorem shows an upper bound for this family of inputs; it also provides motivation for a new algorithm that does not push out packets that are currently being processed. This restriction is practical (if a packet is being processed, perhaps this means that it has left the queue and gone on, e.g., to CPU cache), and the analysis of such an algorithm is an interesting problem that we leave open.

Theorem 6. *For every $B > 0$ and $k > 0$, if* LPO *never pushes out packets that are currently being processed then* LPO *is at most $\left(\log_2 k + 3 + \frac{B-1}{B}\right)$-competitive.*

Proof. The case when there is no congestion during iteration is identical to the same case of Theorem 5.

If, during an iteration I, $\mathrm{LPO_p}$'s buffer is congested, it is full and it will remain full till the end of iteration. If during I OPT transmits less than B packets then we are done. Otherwise, consider sub-intervals of time during I when OPT transmits exactly B packets.

We denote by A_i^O the average number of processing passes between all packets transmitted by OPT during the ith subinterval. We also denote by A_i^s and by A_i^e the average number of residual processing passes among all packets in LPO$_{\text{p}}$'s buffer at the start and at the end of the ith subinterval, respectively. Since any packet processed during the subinterval is not pushed out, $A_i^e = \min(A_i, A_i^s - A_i)$. Clearly, as a result the maximal number of subintervals during an iteration is achieved when $A_i = A_i^s - A_i$. Therefore, the maximal number of subintervals during an iteration is bounded by $\log_2 k$ (recall that $A_1^s \le k$). By definition, OPT can gain at most $2B$ packets at the time of the first congestion during iteration. In the worst case, from the end of the last subinterval till the end of iteration OPT can transmit at most $B - 1$ additional packets. Thus, during a congested iteration OPT transmits at most $(\log_2 k + 2)B + B - 1$ packets. Moreover, by Claim 3.2(2), between two subsequent iterations OPT can transmit at most B additional packets. Thus, LPO$_{\text{p}}$ is at most $\left(\log_2 k + 3 + \frac{B-1}{B}\right)$-competitive. $\qquad\square$

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 [7] that a push-out algorithm that processes packets with less required processing first is optimal. In what follows we denote it by OPT*. Clearly, OPT in the FIFO queueing model does not outperform OPT*.

Our traffic is generated using an ON-OFF Markov modulated Poisson process (MMPP), which we use to simulate bursty traffic. The choice of parameters is governed by the 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. For a choice of parameters yielding an average packet arrival rate of λ, where every packet has its required number of passes chosen uniformly at random within the range $[1, k]$, we obtain an average arrival load (in terms of required passes) of $\lambda \cdot \frac{k+1}{2}$.

In our experiments, the "OFF" state has average arrival rate $\lambda = 0.3$, and the "ON" state has average arrival rate $\lambda = 4.5$ (the number of packets is uniformly distributed between 3 and 6). By performing simulations for variable values of the maximal number of required passes k in the range $[1, 40]$, we essentially evaluate the performance of our algorithms in settings ranging from underload (average arrival load of 0.3 for $k = 1$ and 0.6 for $k = 2$) to extreme overload (average arrival load of 180 in the "ON" state for $k = 40$), which enables us to validate the performance of our algorithms in various traffic scenarios.

Figure 2 shows the results of our simulations. The vertical axis in all figures represents the ratio between the algorithm's performance and OPT^* performance given the arrival sequence (so the red line corresponding to OPT* is always horizontal at 1).

We conduct three sets of simulations: the first one is targeted at a better understanding of the dependence on the number of processing cycles, the second evaluates dependency of performance from buffer size, and the third aims to evaluate the power of having multiple cores.

We note that the standard deviation throughout our simulation study never exceeds 0.05 (deviation bars are omitted from the figures for readability). For every choice of parameters, we conducted 200,000 rounds (time slots) of simulation.

4.1 Variable Maximum Number of Required Processing Cycles

In these simulations, we restricted our attention to the single core case ($C = 1$). The top row of graphs on Fig. 2 shows that OPT* keeps outperforming LPO and NPO more and more as k grows. In these settings, the difference in the order of processing between OPT^* and PO is small. The performance of LPO versus NPO degrades moderately since LPO is a push-out algorithm. This behaviour is of course as expected.
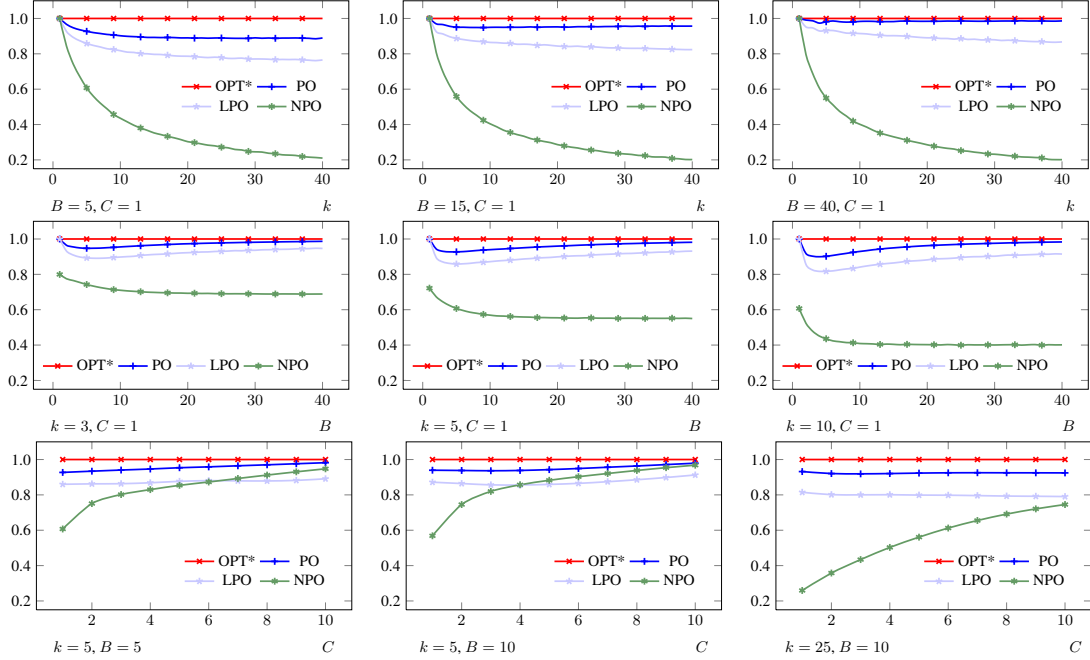
Fig. 2. Competitive ratio as a function of parameters: top row, of k; middle row, of B; bottom row, of C.

4.2 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 middle row on Fig. 2 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).

4.3 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. 2 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

The increasingly heterogeneous needs of NP traffic processing pose novel design challenges for NP architects. 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. 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.

References

1. Xelerated: X11 family of network processors, product brief (2010) [Online] `http://www.xelerated.com/Uploads/Files/67.pdf`.
2. Cavium: Octeon ii cn68xx multi-core mips64 processors, product brief (2010) [Online] `http://www.caviumnetworks.com/OCTEON-II_CN68XX.html`.
3. AMCC: np7310 10 gbps network processor, product brief (2010) [Online] `http://www.appliedmicro.com/MyAMCC/jsp/public/productDetail/product_detail.jsp?productID=nP7310`.
4. Cisco: The cisco quantumflow processor, product brief (2010) [Online] `http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_overview_c22-448936.html`.
5. EZChip: Np-4 network processor, product brief (2010) [Online] `http://www.ezchip.com/p_np4.htm`.
6. McKeown, N., Parulkar, G., Shenker, S., Anderson, T., Peterson, L., Turner, J., Balakrishnan, H., Rexford, J.: Openflow switch specification (2011) [Online] `http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf`.
7. Keslassy, I., Kogan, K., Scalosub, G., Segal, M.: Providing performance guarantees in multipass network processors. In: INFOCOM. (2011) 3191–3199
8. Kogan, K., López-Ortiz, A., Scalosub, G., Segal, M.: Large profits or fast gains: A dilemma in maximizing throughput with applications to network processors (2012) [Online] `http://arxiv.org/abs/1202.5755`.
9. Goldwasser, M.: A survey of buffer management policies for packet switches. SIGACT News **41** (2010) 100–128
10. Mansour, Y., Patt-Shamir, B., Lapid, O.: Optimal smoothing schedules for real-time streams. Distributed Computing **17** (2004) 77–89
11. Kesselman, A., Lotker, Z., Mansour, Y., Patt-Shamir, B., Schieber, B., Sviridenko, M.: Buffer overflow management in QoS switches. SIAM Journal on Computing **33** (2004) 563–583
12. Aiello, W., Mansour, Y., Rajagopalan, S., Rosén, A.: Competitive queue policies for differentiated services. Journal of Algorithms **55** (2005) 113–141
13. Englert, M., Westermann, M.: Lower and upper bounds on FIFO buffer management in QoS switches. Algorithmica **53** (2009) 523–548
14. Kesselman, A., Patt-Shamir, B., Scalosub, G.: Competitive buffer management with packet dependencies. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS). (2009)
15. Mansour, Y., Patt-Shamir, B., Rawitz, D.: Overflow management with multipart packets. In: INFOCOM. (2011) 2606–2614
16. Albers, S., Schmidt, M.: On the performance of greedy algorithms in packet buffering. SIAM Journal on Computing **35** (2005) 278–304
17. Azar, Y., Richter, Y.: An improved algorithm for cioq switches. ACM Transactions on algorithms **2** (2006) 282–295
18. Azar, Y., Litichevskey, A.: Maximizing throughput in multi-queue switches. Algorithmica **45** (2006) 69–90
19. Kesselman, A., Kogan, K., Segal, M.: Improved competitive performance bounds for cioq switches. In: ESA. (2008) 577–588
20. Kesselman, A., Kogan, K., Segal, M.: Packet mode and qos algorithms for buffered crossbar switches with fifo queuing. Distributed Computing **23** (2010) 163–175
21. Albers, S., Jacobs, T.: An experimental study of new and known online packet buffering algorithms. Algorithmica **57** (2010) 725–746
22. Schrage, L.: A proof of the optimality of the shortest remaining processing time discipline. Operations Research **16** (1968) 687–690
23. Leonardi, S., Raz, D.: Approximating total flow time on parallel machines. In: STOC. (1997) 110–119

24. Muthukrishnan, S., Rajaraman, R., Shaheen, A., Gehrke, J.E.: Online scheduling to minimize average stretch. SIAM Journal on Computing **34** (2005) 433–452
25. Motwani, R., Phillips, S., Torng, E.: Non-clairvoyant scheduling. Theoretical Computer Science **130** (1994) 17–47
26. Pruhs, K.: Competitive online scheduling for server systems. SIGMETRICS Performance Evaluation Review **34** (2007) 52–58
27. Brucker, P., Heitmann, S., Hurink, J., Nieberg, T.: Job-shop scheduling with limited capacity buffers. OR Spectrum **28** (2006) 151–176
28. Ruiz, R., Vázquez-Rodrígue, J.A.: The hybrid flow shop scheduling problem. European Journal of Operational Research **205** (2010) 1–18
29. Wolf, T., Pappu, P., Franklin, M.A.: Predictive scheduling of network processors. Computer Networks **41** (2003) 601–621
30. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Communications of the ACM **28** (1985) 202–208
31. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press (1998)

Appendix

Proof (of Theorem 2). (*1*) *Lower bound.* Assume for simplicity that B/C is an integer. Consider the following set of arrivals. During the first time slot arrive $B - C$ packets with maximal number of passes k. From the same time slot an iteration is started. During each iteration C packets with k processing cycles arrive. Since NPO is greedy it accepts all of them and its buffer is full. OPT does not accept these packets. During the arrival phase of the next kC time slots the buffer of NPO is full since it is non-push-out and implements FIFO order. During this time interval arrive C packets with a single processing cycle each time slot. During each iteration OPT transmits kC packets but NPO transmits only C packets. In contrast to the other iterations during the last time slot of the last iteration a burst of B packets arrives. So at the end of the arrival sequence both buffers are full. Thus, the competitiveness of NPO is at least $\frac{ikC+B}{iC+B}$, $i \geq 1$. Therefore, for sufficiently big value of k, NPO is at least k-competitive.

(*2*) *Upper bound.* Observe that NPO must fill up its buffer before it drops any packets. Moreover, so long as the NPO buffer is not empty then 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 transmitted at most k packets. Hence, the number of transmitted packets at time t for NPO is at least $\lfloor t/k \rfloor$ while OPT transmitted at most t packets for a competitive ratio of k so long as the NPO buffer did not become empty before OPT's did.

If, on the other hand, NPO empties its buffer first, this means 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 transmitted at most B packets.

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

Proof (of Theorem 3). Case 1. $k \geq B$. In this case, the same hard instance works for both PO and LPO. 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	$\mathrm{IB}_t^{\{\mathrm{PO},\mathrm{LPO}\}}$	# {PO, LPO}	$\mathrm{IB}_t^{\mathrm{OPT}}$	# OPT
1	1 B	1 B	0	1	1
2	1	1 1 B-1	0	1	2
3	1	1 1 1 B-2	0	1	3
...		...		...	
$B - 2$	1	1 ... 1 2	0	1	$B - 2$
$B - 1$	1 × B	1 ... 1 1	1	1 ... 1 1	$B - 1$
...		...		...	
$2B - 1$			B		$2B - 2$

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.

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

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 $\alpha B(1 + \frac{1}{k})$, 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

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

Solving for α, we get $\alpha = 1 - \frac{1}{k}$. During the sequence, OPT has processed $\alpha B \left(1 + \frac{1}{k} + \frac{1}{k^2} + \dots\right) + B = 2B$ 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, while PO has processed $\frac{\alpha B}{k}$ packets, so it has $\frac{\alpha B}{k}$ free spaces in the queue. 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 1. 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

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

and 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. $\square$

Proof (of Theorem 4). We proceed by induction on B. For the induction base, we begin with the basic construction that works for $k = \Omega(B^2)$.

Lemma 3. *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 exactly* 3.

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

$$\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}\, \dots \, \boxed{\text{B-1}}\,\boxed{\text{(B-1)(B-2)}}.$$

14

Over the next $(B-1)(B-2)$ 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 $(B-1)(B-2)$ steps OPT's queue is empty, and PO's (LPO's) queue looks like

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

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 $(B-1)(B-2)+B$ 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, for the case of unlimited size incoming burst 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. $\qquad\square$

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 $n - O(\frac{1}{B})$, and the construction requires maximal required work per packet less than $S = \Omega(B^{n-1})$.

Let us now use the construction from Lemma 3, but add S to every packet's required work and, consequently, $S(B-1)$ to the first packet's required work:

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

At first (for the first $(B-1)(B-2+S)$ 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

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

Now we can add packets from the previous construction (one by one in the unit-size burst case or 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, we arrive at 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. $\qquad\square$