Multi-Queued Network Processors for Packets with Heterogeneous Processing Requirements

Kirill Kogan Alejandro López-Ortiz
School of Computer Science
University of Waterloo
{kkogan, alopez-o}@uwaterloo.ca

Sergey I. Nikolenko
Steklov Mathematical Institute
St. Petersburg Academic University
St. Petersburg, Russia
sergey@logic.pdmi.ras.ru

Alexander V. Sirotkin
St. Petersburg Institute for
Informatics and Automation
St. Petersburg Academic University
St. Petersburg, Russia
alexander.sirotkin@gmail.com

***Abstract*—Modern network processors (NPs) increasingly deal with packets with heterogeneous processing requirements. In this work, we consider the fundamental problem of managing a bounded size buffer at the input queue of an NP. Incoming traffic consists of packets, each packet requiring several rounds of processing before it can be transmitted out of the queue. The objective is to maximize the total number of successfully transmitted packets. In such an environment, it is well known that Shortest-Remaining-Processing-Time (SRPT) first scheduling with push-out is optimal [1]. However, it is hard to implement both priority queueing (PQ) by remaining processing and the push-out mechanism simultaneously in an NP. We explore alternatives for this architecture, addressing the simplicity vs. performance system design tradeoffs. We design a simplified architecture and provide worst-case guarantees for its throughput performance in different settings. We also conduct a comprehensive simulation study that validates our results.**

Keywords: scheduling, buffer management, routers, online algorithms, competitive analysis.

I. Introduction

Network Processors (NPs) are widely used to perform packet processing tasks in modern high-speed routers. These architectures are efficient for simple traffic profiles. However, modern intelligent networking requires sophisticated features such as advanced VPN services, deep packet inspection, firewalls and intrusion detection, to name just a few [2]–[5].

Processing requirements can be approximately predicted on an NP for a given configuration [6]. Moreover, in OpenFlow [7] reactive configuration mode required processing can be estimated by the controller.

A buffer management policy plays a significant role in an NP architecture to guarantee that packets with long processing do not monopolize the processor. One property that has significant impact on the performance of buffer management policies is the processing order of packets ("run-for-completion", processing with recycles etc. [1]). Another important property of a buffer management policy is its ability to push out already admitted packets. This can have an even stronger effect on performance than the choice of processing order. It is well known that, in the case of a single queue, a greedy push-out policy PQ that processes a packet with minimal required processing first is optimal with respect to throughput [1]. To understand the impact of push-out on the throughput performance, recall that a greedy push-out policy that implements FIFO or FIFO with recycles processing order is at least $\Omega(\log k)$-competitive versus the optimal clairvoyant algorithm [1], [8] where k is the maximal possible required processing. At the same time, non-push-out policies with the same processing orders have poor (k-competitive) performance in the worst case [1], [8]. However, it is a complex task from both a logical and system point of view to implement both an advanced processing order and a push-out mechanism simultaneously.

978-1-4673-5494-3/13/$31.00 © 2013 IEEE

A. Our Approach and Contributions

The first interesting question that we study is to find a simplified architecture that has, on one hand, comparable throughput guarantees to a single queue architecture that implements PQ order but, on the other hand, does not require implementation of the push-out mechanism and advanced processing order. As an example of such an alternative, we consider a multi-queued system where each queue accepts packets with a given processing requirement, and the total size of all queue buffers is the same as the buffer size in the single queue architecture (see Fig. 1). One major advantage of such a system is a significantly simpler implementation: since each queue holds packets with the same processing requirement, there is no need for a push-out mechanism, and a given queue simply processes packets in FIFO order. This

separation provides an additional opportunity to allocate different processing bandwidths to packets from different queues in order to implement more sophisticated QoS mechanisms; admission control can be applied to each queue separately, in a fully distributed manner. Moreover, if queues are implemented as separate buffers then total memory bandwidth increases proportionally to the number of queues.

An obvious drawback of this architecture is the preconfigured limit on the size of each queue, so that buffers are not shared between packets with different processing requirements. However, in many applications this is not a limitation but rather an additional advantage since it is desirable to limit the number of packets from the same class. In this work, we explore the effect of such system simplifications on performance. Our contributions include: (1) a new queuing architecture for packets with heterogeneous processing, (2) performance analysis for this system, and (3) a simulation study to validate our conclusions.

Once we have introduced a simplified multi-queue architecture, the next step is to design a buffer management policy with the best possible performance. At first, we concentrate on throughput maximization. The first question is to understand the impact of two important characteristics: buffer occupancy and residual work. It is unclear which one is more significant: if one policy processes packets from the longest queue while another processes packets with minimal residual work, which is better? We tackle this question in Section II-A. Once this property is clear, we define a simple greedy algorithm that optimizes this property and evaluate system performance. As a result, we estimate competitive ratios and the value of speedup needed to achieve optimal throughput. We demonstrate that a moderate speedup between $\frac{3}{2}$ and 2 suffices to obtain results on par with the optimal single queue case. For systems with only two kinds of processing requirements, 1 and k (this is a special case that often occurs in practice), the proposed algorithm has near optimal performance.

The multi-queue architecture is more flexible than the original single queue-case; it yields a straightforward implementation of the fairness property between different types of traffic identified by residual work. We consider an implementation of fairness on two different levels: packet-based and cycle-based. We demonstrate that fairness does have its price, and in Section II-D, we estimate the impact of fairness on throughput. In Section III, we demonstrate that none of the proposed reasonable policies are comparable in the worst case.

Although in Section IV we conduct a comprehensive simulation study to validate our results, the main emphasis of this work is on worst-case performance guarantees for the introduced architecture. Our results provide performance guarantees that are independent of the incoming traffic distribution and its processing requirements, which is desirable since properties of heterogeneous processing often differ significantly between various locations in the network, and there are no universal stochastic assumptions to be made about it. An additional advantage of the worst-case approach is that we are able to estimate the required speedup factor (in our case number of cores) to guarantee performance close to optimal for any type of arrivals.

B. Related Work

Keslassy et al. [1] were the first to consider a single queue buffer management and scheduling in the context of network processors with heterogeneous processing requirements for arriving traffic. They studied both PQ (Priority Queue) and FIFO schedulers with recycles, in both the push-out and the non-push-out buffer management cases, where a packet is recycled after processing according to the priority policy. For the case of FIFO with recycles, only preliminary results were obtained. Kogan et al. [8] considered the FIFO case for packets with heterogeneous processing and proved several upper and lower bounds for the proposed algorithms.

Our current work can be viewed as part of a larger research effort concentrated on studying competitive algorithms for buffer management of bounded buffers. A recent survey by Goldwasser [9] provides an excellent overview of this field. This line of research, initiated in [10], [11], has received a colossal attention over the past decade.

Various models have been proposed and studied, including QoS-oriented models where packets have weights [10]–[13] and models where packets have dependencies [14], [15]. A related field that has recently attracted much attention focuses on various switch architectures and aims to design competitive algorithms for such multi-queued scenarios; see, e.g., [16]–[20]. However, these models do not cover the case of packets with heterogeneous processing requirements.

Pruhs [21] provides a comprehensive overview of competitive online scheduling for server systems. Note that scheduling for server systems is mostly concentrated on average response time, but we focus mostly on the throughput estimation. Moreover, scheduling of server systems does not allow jobs to be dropped, which is an inherent aspect of our model due to limited-size buffer.

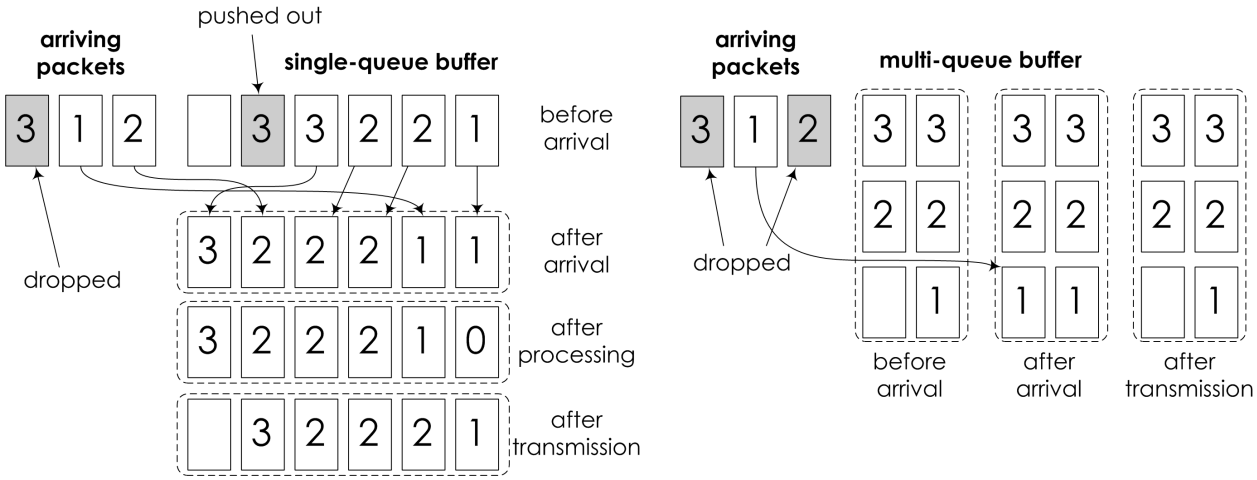

Fig. 1. A sample time slot. On the left: a single priority queue with buffer of size $B = 6$; on the right: a multi-queued switch with three queues (here $k = 3$) and buffer of size $B = 2$ each. Dashed lines enclose buffers. Note that the sets of packets in the buffers at the end of the time slot are different.

C. Model Description

Consider a system with k separate queues, each queue holding packets with the same initial required processing. In what follows we assume that all queues are of the same size B. For the case of a single core NP architecture, we consider $k \times 1$ input-queued switch. For the multi-core case, we consider the same $k \times 1$ input-queued switch with a speedup of S, i.e., with S general purpose cores that can process packets from any one of k queues.

We consider a model with arbitrary packet arrival, i.e., incoming traffic is not governed by any specific stochastic process and may be adversarial. Arrivals come in a sequence of unit-sized packets. Each arriving packet p is labelled by the number of required processing cycles $r(p) \in \{1, \ldots, k\}$ that defines the corresponding input queue. This number is essentially the number of times the packet should be processed before it can be successfully transmitted. In practice, $r(p)$ may be an approximation, or may become known after the first pass rather than from the start.

In what follows, we adopt the terminology used in [22]. A policy performs two main tasks, namely *buffer management*, which handles admission control of newly arrived packets, and *scheduling* that chooses a queue whose *head of line* (HOL) packet will be processed next. We assume discrete slotted time, each time slot t consisting of three phases: (i) *arrival*: new packets arrive, and the buffer management unit performs admission control; (ii) *scheduling*: a packet is processed according to a scheduling policy; (iii) *transmission*: packets with zero required processing are transmitted and leave the queue. In case of nontrivial speedup $S > 1$, scheduling and transmission run S times per time slot.

Note that unlike the optimal single-queued architecture, a packet may be dropped only upon arrival (there is no push-out). A packet contributes one unit to the objective function when it is successfully transmitted. The goal is to devise buffer management algorithms that maximize the overall throughput, i.e., the total number of packets transmitted from the queue.

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.

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. The required processing of an already transmitted packet is zero. A packet p is *better* than packet q at time t if $r_t(p) \leq r_t(q)$. We denote by $r(p)$ the required work of a packet p during arrival. In what follows, we denote by $\boxed{m}$ a packet that requires $m \leq k$ processing cycles; by $l \times \boxed{m}$, a burst of l packets where each one requires m processing cycles.

In this work, we do not assume any specific traffic distribution but rather analyze our switching policies against adversarial traffic using competitive analysis [23], [24], which provides a uniform throughput guarantee for all traffic patterns. An online algorithm A is said to be α-*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.

II. Price of Simplicity

In this section we evaluate the price of simplicity. The multi-queue setting has no push-out mechanism and does not have to sort packets by required processing – so how much worse is it? Here we concentrate on maximizing throughput performance and ignore other additional important properties such as fairness; we will return to them later: the impact of fairness on throughput performance will be evaluated in Section II-D.

A. Throughput Maximization

In this section, we analyze the impact of different characteristics on throughput maximization. Once the "dominant" characteristic is found, we define a simple greedy algorithm that optimizes this characteristic and evaluate system performance in these settings from the perspective of worst-case performance under adversarial traffic. We identify two major characteristics that can potentially affect throughput performance: buffer occupancy and required processing.

First, we concentrate on the buffer occupancy characteristic and propose the Longest-Queue-First (LQF) online policy that proactively processes the HOL packet of a currently longest queue to accommodate with the future potential incoming burst. Note that a *lower* bound on the competitive ratio is *bad* news for the policy: it means that it is that much worse than the optimal; conversely, an *upper* bound shows that the policy is at least that *good*.

Theorem 1: LQF is at least $\frac{m}{2}$-competitive for $m = \min\{k, B\}$.

Proof: In this case, in the first burst we send $m \times \boxed{m}$ and $m \times \boxed{1}$. Over the next $m+1$ steps, LQF processes 2 packets (one $\boxed{1}$ and one $\boxed{m}$) while OPT processes m packets from queue 1. Then the burst is repeated. ■

Theorem 1 demonstrates that a processing order that is based on buffer occupancy statistics does not really help in bursty traffic. Next, we introduce a different policy that also uses the buffer occupancy characteristic but processes a packet from a currently shortest queue. We call this policy Shortest-Queue-First (SQF).

Theorem 2: SQF is at least k-competitive.

Proof: First burst: $1 \times \boxed{k}$ and $B \times \boxed{1}$; then every step one more $\boxed{1}$ arrives, and every k steps one more $\boxed{k}$ arrives. Thus, SQF will always be processing queue k with one packet since it is always the shortest while OPT is free to process queue 1 on every time slot. ■

Theorems 1 and 2 demonstrate that there is little difference in the worst case between policies based on the buffer occupancy characteristic and a policy that processes any available packet first (note that any reasonable policy has a trivial upper bound of k since as long as packets are available, it is able to process at least one packet per k time slots). Next, we consider the second characteristic and introduce another policy that, in contrast to the previous ones, does not care about the current buffer occupancy and processes a packet from a non-empty queue with minimal required processing. We call it Minimal-Queue-First (MQF). For the lower bound, note that MQF may spend substantial effort processing a queue which will never be congested.

Theorem 3: MQF is at least $\left(1 + \frac{k-1}{2k}\right)$-competitive.

Proof: During the first time slot, there arrive $B \times \boxed{k-1}$ and $B \times \boxed{k}$. Over the next $B(k-1)$ steps, MQF processes all packets from queue $k-1$ while OPT processes $\frac{k-1}{k}B$ packets from queue k. Then the second burst arrives with $B \times \boxed{k}$, filling queue k for both algorithms; then they both are allotted time enough to flush out. As a result, MQF has processed $2B$ packets while OPT has processed $2B + \frac{k-1}{k}B$ packets, and this sequence can be repeated to get the bound. ■

Theorem 3 demonstrates that processing requirement potentially has a greater impact on the throughput than buffer occupancy. In the next section, we prove an upper bound, namely we demonstrate that MQF is at most 2-competitive. Thus, we will show that even moderate speedup in the (1,2] range can guarantee almost optimal throughput for a multi-queued architecture with the MQF policy for heterogeneous packet processing.

In Section II-C, we consider a special case with only two queues. This special case is interesting by itself: one queue (with one kind of required processing) may correspond to "commodity" packets processed to completion while the second queue may hold packets with "advanced" features that require k cycles. We prove a tight bound on MQF competitiveness in this case.

The opposite of MQF, a policy MaxQF that processes maximal valued queue first, is obviously the worst policy among those we consider in this work.

Theorem 4: MaxQF is at least k-competitive.

Proof: Fill queues 1 and k and keep them full. MaxQF will always process $\boxed{k}$s while OPT can concentrate on $\boxed{1}$s. ■

B. Upper Bound on the Competitiveness of MQF

Theorem 5: MQF is at most 2-competitive.

To prove the main result we introduce the notion of an *iteration*. An iteration is a time interval between two

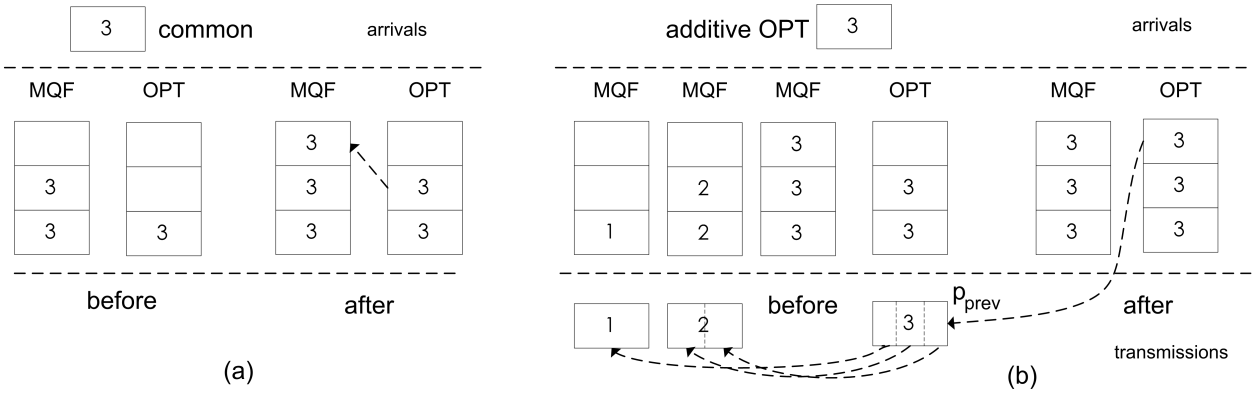

Fig. 2. Mapping of OPT fractions to MQF fractions. The sub-figure (a) demonstrates a mapping of a common packet to itself. The sub-figure (b) demonstrates a mapping of an additive OPT packet to p_{prev} that is mapped to MQF fractions. We are to show that a size of each such p_{prev} fraction is at most the size of a mapped MQF fraction and at the end of an iteration all p_{prev} packets are transmitted.

closest time slots s and e such that at the beginning of time slot s and at the end of time slot e, MQF buffer is empty. We consider the version of OPT that at the beginning of time slot $e+1$ transmits all packets still residing in OPT buffer, with extra gain to OPT's throughput. Denote by T^A the set of packets transmitted by an algorithm A during an iteration. A packet p that is accepted by algorithm A_1 but is not accepted by algorithm A_2 is called *additive* for A_1. A packet that is accepted by both algorithms OPT and MQF is called *common*. The following lemma introduces a constraint on OPT that does not violate optimality.

Lemma 1: There exists an optimal offline algorithm that accepts T^{MQF} during an iteration.

Proof: During an iteration for each additive MQF packet there exists at least one additive OPT packet. Since MQF is choosing for processing a non-empty queue with minimal required processing first, for each additive MQF packet p there is an additive OPT packet q. If $r(p) \leq r(q)$ then such packets can be simply exchanged in OPT's schedule. Otherwise, there are no better arrivals while processing a "heavy" additive MQF packet p which lets MQF complete processing the heavier packet. So in both cases the algorithm that implements the OPT schedule during an iteration after the exchange of such p and q additive packets in MQF and OPT schedules, respectively, remains optimal. ■

From this point on we consider the constrained version of OPT that obeys Lemma 1. To account for the number of accepted OPT packets, we introduce a mapping m from the set of packets admitted by OPT, $\mathcal{P}_{\mathrm{OPT}}$, to the set of packets admitted by MQF, $\mathcal{P}_{\mathrm{MQF}}$. Thus, a common packet p admitted by both OPT and MQF belongs to $\mathcal{P}_{\mathrm{OPT}} \cap \mathcal{P}_{\mathrm{MQF}}$. An additive OPT packet q belongs to $\mathcal{P}_{\mathrm{OPT}} \setminus \mathcal{P}_{\mathrm{MQF}}$; by Lemma 1, there are no additive MQF packets during an iteration.

We map every common packet to itself: $m(p) = p$ for $p \in \mathcal{P}_{OPT} \cap \mathcal{P}_{MQF}$ (see Fig. 2(a)). Now let us consider additive OPT packets. Let b denote the buffer that would hold a packet p if admitted. For the purposes of this analysis we treat each buffer b_i as an unbounded linear array $b_i[1..\infty]$ with a sliding window of size B (buffer capacity). For example, consider buffer b_2 initially holding packets q, q' and q''. Then the window currently spans from position 1 to position B in the array, with the following assignments: $b_2[1] = q$, $b_2[2] = q'$, $b_2[3] = q''$, $b_2[4] = \varnothing$, ..., $b_2[B] = \varnothing$. Now if a packet q is processed and transmitted, the current window over b_2 covers $2, \ldots B+1$ with $B+1$ initialized as empty.

Recall that MQF has a greedy admission policy, so the only way in which an additive OPT packet p is created is when p arrives and b is full under the MQF policy, while b has available space under the OPT scheduling policy. We denote by b_t^{MQF} and b_t^{OPT} the corresponding state of b under each of the two policies respectively at time t. If the value of t is clear we omit the subscript.

Since MQF's buffer is full while OPT is not, there is only one possible reason for this discrepancy: OPT processed packets in b ahead of MQF in the schedule.

Let i denote the cell in b^{OPT} into which the additive packet p is admitted to. Now we focus on the packet p_{prev} in position $b^{\mathrm{OPT}}[i-B]$. Informally, we can think of p_{prev} as the packet that had to be processed and transmitted out by OPT in order to make space for the additive packet p freshly admitted into b^{OPT}.[1]

For this case we extend the mapping m to a fractional

[1]Note, however, that by the time p arrives b^{OPT} might well have more than one available cell.

mapping in which a given packet in $\mathcal{P}_{\mathrm{OPT}}$ can be partitioned into segments, with each part being mapped to a fractional part of a packet in $\mathcal{P}_{\mathrm{MQF}}$. We then will show that the sum of the fractional pre-images of a packet $q \in \mathcal{P}_{\mathrm{MQF}}$ is at most 1 and that, furthermore, over the entire iteration the number of common packets transmitted by MQF is equal to or larger than the number of additive packets created by OPT during iteration. This plus the packet from the identity mapping of common packets gives a competitive ratio of two.

The fractional mapping works as follows.

Let $r := r(p_{\mathrm{prev}})$. For every time slot when p_{prev} was being processed by OPT we map a $1/r$ fraction of p_{prev} to a $1/r(q)$ fraction of the packet q that was being processed by MQF at exactly that time (Fig. 2(b)). We have to prove that q always exists. By Lemma 1, at most $B-1$ packets have been admitted to b^{OPT} since OPT processed p_{prev}. Hence, at the time p_{prev} was processed by OPT, b^{MQF} must have contained at least one packet as otherwise b^{MQF} could not presently be full. Moreover, at least one of these packets, termed q_{prev}, is still presently in b^{MQF}. This means that for every time interval when p_{prev} was being processed MQF had q_{prev} in its buffer. Furthermore, by its greedy nature, at that time MQF was processing packets $q_1, q_2, \ldots q_r$ (not necessarily distinct) with required work $r(q_i)$ no larger than r as otherwise MQF would have processed q_{prev}; equivalently, $1/r \le 1/r(q_i)$ for all $1 \le i \le r$.

To finish the proof it remains to count how many additive packets in b^{OPT} were mapped to how many common packets in b^{MQF}. The first number is obtained by adding the fractional components of additive packets in b^{OPT} while the second corresponds to the sum of their fractional images in b^{MQF}. Let N be a number of additive OPT packets that are created during an iteration, $|\mathcal{P}_{\mathrm{OPT}} - \mathcal{P}_{\mathrm{MQF}}|$. Then $N = \sum_{i=1}^{W} 1/r_i \le \sum_{i=1}^{W} 1/r(q_i) \le |\mathcal{P}_{\mathrm{MQF}}|$, where W is the total work by OPT on additive packets.

Since all p_{prev} packets are transmitted at the end of an iteration and assigned MQF fractions are bigger or have the same size as the mapped OPT mates, the 2-competitiveness of MQF follows. This result holds for any value of speedup $S \ge 1$ since the mapping routine does not depend on S.

C. Tight Bounds for Two-Queued MQF

In this section, we study the special case when there are only two types of packets, $\boxed{a}$ and $\boxed{b}$, corresponding to only two queues in the system. For this special case, we show exactly matching lower and upper bounds.

In what follows, we assume that $a < b$ and call smaller packets a-packets and larger packets b-packets. We call an *iteration* a period of time between two time slots during which the MQF buffer is empty.

We begin with the lower bound that gives an enlightening example for the upper bound's proof.

Theorem 6: The competitiveness of MQF with two queues is at least

$$\left(1 + \frac{1 + \left\lfloor \frac{aB-1}{b} \right\rfloor}{B + \left\lceil \frac{1}{a}\left(b\left\lfloor \frac{aB-1}{b} \right\rfloor + 1\right)\right\rceil}\right).$$

Proof: We denote $A = \left\lceil \frac{1}{a}\left(b\left\lfloor \frac{aB-1}{b} \right\rfloor + 1\right)\right\rceil$. First burst: $B \times \boxed{b}$ and $A \times \boxed{a}$. In time Aa, MQF has processed all a-packets while OPT has processed at least $\left\lfloor \frac{aB-1}{b} \right\rfloor$ b-packets and spent one extra unit of work on the next b-packet. We now wait for $b-1$ more time steps so OPT processes one more b-packet while MQF gets its first b-packet down to $\boxed{1}$. Then we send enough packets to fill both buffers in both algorithms and wait for them to flush out, obtaining the bound. ∎

Now we show that this bound is tight.

Theorem 7: The competitiveness of MQF with two queues is at most

$$\left(1 + \frac{1 + \left\lfloor \frac{aB-1}{b} \right\rfloor}{B + \left\lceil \frac{1}{a}\left(b\left\lfloor \frac{aB-1}{b} \right\rfloor + 1\right)\right\rceil}\right).$$

Lemma 2: At any time moment, MQF has processed at least as many a-packets as OPT.

Proof: Straightforward: MQF gives top priority to a-packets, so if OPT has not dropped an a-packet, MQF has not dropped it either. ∎

Lemma 3: At any time moment during an iteration (i.e., when MQF buffer is not empty), MQF has processed at least as many packets as OPT in this iteration.

Proof: During an iteration, MQF is never idle; therefore, Lemma 2 implies that it processes at least as many packets as OPT: the total work of MQF is at least the same as OPT's, and MQF has processed at least as many smaller packets. ∎

Note that Lemma 3 does not imply that MQF is optimal: after an iteration OPT buffer may be nonempty, so OPT wins the remaining packets over MQF.

Lemma 4: The total number of a-packets processed by OPT and left in OPT's buffer at the end of an iteration equals the total number of a-packets processed by MQF during this iteration.

Proof: Obviously, OPT does not drop a-packets if it can avoid it, otherwise we could substitute processing a b-packet with the dropped a-packet and thus get a policy with the same number of packets but less work. ∎

Proof of Theorem 7: First of all, if queue b is never congested during an iteration, MQF works optimally,

and we are done. Hence, we consider an iteration during which at least B packets have entered queue b.

We denote by A the number of a-packets MQF has processed over an iteration; by Lemma 2, OPT has also processed no more than A a-packets (in total during the iteration and left over after the iteration). We denote by D the total number of b-packets processed by MQF. If the total number of b-packets processed by OPT and left over in OPT's buffer is less than the number of b-packets processed by MQF then OPT is worse than MQF which is impossible; thus, this OPT's number is at least D, and we denote by E the number of extra b-packets that OPT has.

The work spent by OPT on extra b-packets cannot exceed the work spent by MQF on a-packets, i.e., aA, plus possibly some extra work that has not led to processing any more packets; this extra work does not exceed $b-1$ (in Theorem 6, we have shown where this extra work can actually come from): $aA+b-1 \geq bE$.

Note that if $A > B$ then OPT has had to spend time on at least $(A-B)$ a-packets during the iteration (see Lemma 4), so OPTs extra work in this case does not exceed aB, and

$$E \leq \left\lfloor \frac{aB+b-1}{b} \right\rfloor = 1 + \left\lfloor \frac{aB-1}{b} \right\rfloor .$$

After the iteration is over and OPT has flushed its buffer (we assume that OPT processes all leftover packets before the next iteration), MQF has processed $A+D$ packets while OPT has processed at most $A+D+E$ packets, for a ratio of $1+\frac{E}{A+D}$. Since $D \geq B$,

$$\frac{E}{A+D} \leq \frac{E}{A+B} \leq \frac{E}{B+\left\lceil \frac{1}{a}\left(bE-b+1\right)\right\rceil},$$

and this fraction is monotonically increasing as E increases so we can substitute the upper bound for E. ■

Observation 1: For systems with only two kinds of packets $\boxed{1}$ and $\boxed{k}$ (this is a special case that often occurs in practice), our bound simplifies to

$$1 + \frac{1+\left\lfloor \frac{B-1}{k} \right\rfloor}{B+1+\left\lceil \left\lfloor \frac{B-1}{k} \right\rfloor k \right\rceil},$$

which exactly equals $1+\frac{1}{B+1}$ for $k \geq B$ and approximately equals $1+\frac{1}{2k}$ for $B \gg k$.

Note that this reasoning explains why MQF is so good in simulations (Section IV): it is highly unlikely that an iteration actually ends with a nonempty OPT buffer thus allowing OPT to realize its advantage in a simulation, while during an iteration MQF is as good as OPT.

D. Fairness Implementation

In this part we evaluate the impact of implementation of fairness on performance of throughput. We consider fairness for both packet and processing cycle levels. Since LQF is at least $\frac{m}{2}$-competitive, where $m = \min\{k, B\}$, we do not expect to get better throughput performance for policies that implement the fairness property on any level. First, we try to implement fairness on the level of a single packet. The Packet-Round-Robin policy (PRR) implements a round-robin scheme between currently active queues on the packet level by switching to the next active queue after processing the HOL packet in the current queue.

Theorem 8: PRR is at least $\frac{3k(k+2)}{4k+16}$-competitive.

Proof: For PRR, we fill up queues $1, \frac{k}{2}, \frac{k}{2}+1, \ldots, k$. In $\frac{3}{2}\frac{k}{2}\left(\frac{k}{2}+1\right)+1$ steps, PRR will have processed $\frac{k}{2}+2$ packets while OPT processes $\boxed{1}$ on every step, and the process repeats, getting the bound. ■

Although the result is predictable, it is interesting that PRR can outperform SQF. Next, consider the Cycle-Round-Robin policy (CRR) with a round-robin scheme between active queues that switches to the next active queue after each processing cycle.

Theorem 9: CRR is at least $\frac{k}{H(k)}$-competitive for $H(k) = \sum_{i=1}^{k} \frac{1}{i} \approx \ln k + \gamma$.

Proof: For round-robin policies, the worst case is when all relevant queues are always full (complete congestion). In this case, over $k!k$ steps CRR spends exactly $k!$ processing steps on each queue and thus processes $\sum_{i=1}^{k} \frac{k!}{i}$ packets, while OPT can concentrate on queue 1, processing $k!k$ packets in total. ■

The interesting result here is that "fair" CRR can actually outperform LQF that proactively tries to reduce potential future congestion.

III. Worst-Case Incomparabilities

In this section we compare the proposed policies in the worst case. An algorithm A_1 *outperforms* an algorithm A_2 if for any input σ, A_1 transmits at least the number of packets that A_2 transmits. Otherwise, A_1 and A_2 are *incomparable* in the worst case. It turns out that virtually all meaningful policies are incomparable with each other in the worst case. We show that (somewhat counterintuitively) MQF, while it does have a very good upper bound, can in fact in certain situations behave worse than any other policy in this paper.

Theorem 10: MQF is incomparable with SQF, LQF, CRR, and PRR.

Proof: In one direction, incomparability results follow from Theorem 5 since lower competitiveness bounds for other policies are worse than the MQF upper bound

of 2. Next, we show counterexamples for which MQF performs worse than simpler policies.

SQF $>$ MQF. First burst: $B \times \boxed{k-1}$ and $(B-1) \times \boxed{k}$. Thus, SQF processes packets from queue k while MQF processes packets from queue $k-1$. In $\min\{k(B-1), B(k-1)\}$ timeslots, one of these queues will be empty, the second burst follows with $B \times \boxed{k}$, and then both algorithms are allotted time to flush out their buffers. As a result, MQF has processed $2B$ packets while SQF has processed $2B + \min\left\{B-1, B\frac{k-1}{k}\right\}$, which is $\left(1 + \frac{1}{2}\min\left\{\frac{k-1}{k}, \frac{B-1}{B}\right\}\right)$ times better.

LQF $>$ MQF, CRR $>$ MQF, PRR $>$ MQF. All three bounds result from the same set of packets. First burst: $B \times \boxed{k-1}$ and $B \times \boxed{k}$. After $(k-1)B$ packets, send a second burst of $B \times \boxed{k}$ and then let the algorithms flush; thus, MQF will have processed $2B$ packets, and the advantage of other policies is exactly the number of $\boxed{k}$ packets that have been processed. In this case, both LQF and PRR get one $\boxed{k}$ per $2k-1$ time slots, thus they fare $\left(1 + \frac{k-1}{2(2k-1)}\right)$ times better than MQF; CRR gets one $\boxed{k}$ per $2k$ time slots so it is $\left(1 + \frac{k-1}{4k}\right)$ times better than MQF. ■

Next we show the remaining incomparabilities.

Theorem 11: (1) PRR is incomparable with SQF, LQF, and CRR; (2) CRR is incomparable with SQF and LQF; (3) SQF and LQF are incomparable.

Proof: (1) The same example as in Theorem 8 immediately shows that PRR can do worse than SQF and CRR (they perform better on PRR's lower bound). To show that PRR $<$ LQF, send $(B-1) \times \boxed{k}$ and $B \times \boxed{1}$, then keep sending one $\boxed{1}$ per time slot, and let both algorithms empty out their buffers when PRR is out of $\boxed{k}$s. For PRR $>$ SQF, send $(B-1) \times \boxed{k}$ and $B \times \boxed{1}$, keep filling up $\boxed{1}$s and let both algorithms empty buffers when SQF is out of $\boxed{k}$s. For PRR $>$ CRR, first send $B \times \boxed{1}$ and $B \times \boxed{2}$. In $2B$ ticks, CRR will be out of $\boxed{1}$s, having processed $\frac{B}{2} \times \boxed{2}$, and PRR will have processed $\frac{2B}{3} \times \boxed{2}$. Now fill up $\boxed{2}$s and let both algorithms empty their buffers, getting a constant improvement for PRR over CRR. For PRR $>$ LQF, send $B \times \boxed{k}$ and $(B-1) \times \boxed{1}$ and keep adding $\boxed{k}$s on every time slot. As a result, LQF keeps processing $\boxed{k}$ while PRR switches between $\boxed{k}$ and $\boxed{1}$, getting an advantage. (2) Incompatibility results for CRR follow from exactly the same constructions as PRR. (3) For SQF $>$ LQF, send $(B-1) \times \boxed{1}$ and $B \times \boxed{k}$, adding more $\boxed{k}$s as needed. For LQF $>$ SQF, send $B \times \boxed{1}$ and $(B-1) \times \boxed{k}$, adding more $\boxed{1}$s as needed. ■

IV. Simulation Study

In this section, we report on a simulation study that we have conducted in order to compare multi-queued policies in a practical situation. In our simulations, we take the greedy push-out PQ algorithm with a single buffer of size kB as optimal; it is optimal even in its own class [1] and thus obviously performs no worse than the optimal offline algorithm for multi-queued architecture with k queues of size B.

We do not consider real traffic traces for two reasons: first, incoming traffic impacts only a number of arrivals per time slot that are further redistributed dependent on the internal router configuration. Second, it is unclear what time slot size should be chosen; its value will define traffic burstiness even for fixed arrivals and a router configuration. Because of this uncertainty, we evaluate the performance of the proposed architecture on traffic generated synthetically, with an ON-OFF Markov modulated Poisson process (MMPP). 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 the packets. For a Poisson arrival process with intensity λ, and the packets' required processing chosen uniformly at random from $1..k$, we obtain an average arrival load (in terms of required passes) of $\lambda\frac{k+1}{2}$. Thus, in our experiments incoming traffic intensity is governed by: λ_{OFF}, intensity of the Poisson arrival process in the "OFF" state; λ_{ON}, intensity in the "ON" state; k, the maximal number of passes per packet; switching probabilities from "OFF" to "ON" and vice versa (in all experiments, we have set $p(\mathrm{OFF} \to \mathrm{ON}) = 0.1$ and $p(\mathrm{ON} \to \mathrm{OFF}) = 0.25$). We have experimented with $\lambda_{\mathrm{OFF}} \in [0.2,2]$, $\lambda_{\mathrm{ON}} \in [2.5,4]$, and $k \in [1,40]$, thus ranging from severe underload (average arrival load of 0.2) to extreme overload (average arrival load of 1600 in the "ON" state), testing our algorithms in various traffic scenarios. We have also experimented with different values of speedup S (which, if a whole number, is equivalent to the number of cores in the system).

Fig. 3 shows simulation results. The y-axis in all figures represents the ratio between an algorithm's performance and the performance of single-queue PQ policy with buffer size kB (taken as optimal). We have conducted four sets of simulations, varying: (1) maximal number of processing cycles k; (2) buffer size for each queue B; (3) extra speedup, performing S processing steps per time slot; (4) incoming traffic intensity $(\lambda_{\mathrm{OFF}}, \lambda_{\mathrm{ON}})$.

The results are mostly expected: as k grows, most algorithms fall behind the optimal since they lose more

and more time for large packets, except for MQF that is able to stay on small packets, thus showing price of fairness (see Section II-D). As the buffer size B grows, there is little difference in competitiveness estimates, all policies improve at the same rate. As S grows, all algorithms become closer to optimal, which is again expected. On the figures with respect to S, we have also added the graphs $\mathrm{OPT}_{2/3}$ and $\mathrm{OPT}_{1/2}$ that indicate the speedup necessary to become optimal: a point of the OPT_α graph with x-coordinate S shows the optimality of OPT with the same parameters but speedup αS. Finally, as the incoming traffic intensity λ grows (the x-axis show λ_{OFF}, and λ_{ON} also increases at a similar rate) we see that round-robin policies (and MaxQF) perform worse and worse while MQF, SQF, and LQF, on the other hand, improve a little; this also illustrates the price of fairness.

V. Conclusion

Increasingly heterogeneous needs of NP traffic processing pose novel design challenges for router architects. In this paper, we have considered a multi-queued architecture that presents a simplified alternative to single queue architectures with advanced processing orders and push-out mechanisms. We have studied the price of simplicity and concentrate on throughput performance maximization. Next, we have shown how implementing fairness affects the throughput. Finally, we have validated our results through a comprehensive simulation study. As future work, it will be interesting to consider alternative architectures and study how various characteristics in different settings impact on system performance.

Acknowledgements

We thank Srinivasan Keshav for many fruitful discussions that have improved the paper significantly. We also thank Isaac Keslassy for his valuable comments.

Work of S.I. Nikolenko and A.V. Sirotkin has been supported by the Russian Presidential Grant Programme for Young Ph.D.'s, grant no. MK-6628.2012.1, Russian Presidential Grant Programme for Leading Scientific Schools grant no. NSh-3229.2012.1, and Russian Fund for Basic Research grants 12-01-00450-a, 11-01-12135-ofi-m-2011, and 11-01-00760-a.

References

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

[2] J. Mudigonda, H. M. Vin, and R. Yavatkar, "A case for data caching in network processors," unpublished manuscript.

[3] Cavium, "OCTEON II CN68XX multi-core MIPS64 processors, product brief," 2010, [Online] http://www.caviumnetworks.com/OCTEON-II_CN68XX.html.

[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] Juniper, "Junos Trio, white paper," 2009, [Online] http://www.juniper.net/us/en/local/pdf/whitepapers/2000331-en.pdf.

[6] T. Wolf, P. Pappu, and M. A. Franklin, "Predictive scheduling of network processors," *Computer Networks*, vol. 41, no. 5, pp. 601–621, 2003.

[7] N. McKeown, G. Parulkar, S. Shenker, T. Anderson, L. Peterson, J. Turner, H. Balakrishnan, and J. Rexford, "OpenFlow switch specification," 2011, [Online] http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[8] K. Kogan, A. López-Ortiz, S. I. Nikolenko, A. V. Sirotkin, and D. Tugaryov, "FIFO queueing policies for packets with heterogeneous processing," 2012, [Online] http://arxiv.org/abs/1204.5443.

[9] M. Goldwasser, "A survey of buffer management policies for packet switches," *SIGACT News*, vol. 41, no. 1, pp. 100–128, 2010.

[10] Y. Mansour, B. Patt-Shamir, and O. Lapid, "Optimal smoothing schedules for real-time streams," *Distributed Computing*, vol. 17, no. 1, pp. 77–89, 2004.

[11] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko, "Buffer overflow management in QoS switches," *SIAM Journal on Computing*, vol. 33, no. 3, pp. 563–583, 2004.

[12] W. Aiello, Y. Mansour, S. Rajagopolan, and A. Rosén, "Competitive queue policies for differentiated services," *J. Algorithms*, vol. 55, no. 2, pp. 113–141, 2005.

[13] M. Englert and M. Westermann, "Lower and upper bounds on FIFO buffer management in QoS switches," *Algorithmica*, vol. 53, no. 4, pp. 523–548, 2009.

[14] A. Kesselman, B. Patt-Shamir, and G. Scalosub, "Competitive buffer management with packet dependencies," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

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

[16] S. Albers and M. Schmidt, "On the performance of greedy algorithms in packet buffering," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 278–304, 2005.

[17] Y. Azar and Y. Richter, "An improved algorithm for CIOQ switches," *ACM Transactions on algorithms*, vol. 2, no. 2, pp. 282–295, 2006.

[18] Y. Azar and A. Litichevskey, "Maximizing throughput in multi-queue switches," *Algorithmica*, vol. 45, no. 1, pp. 69–90, 2006.

[19] A. Kesselman, K. Kogan, and M. Segal, "Improved competitive performance bounds for cioq switches," *Algorithmica*, vol. 63, no. 1-2, pp. 411–424, 2012.

[20] ——, "Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing," *Distributed Computing*, vol. 23, no. 3, pp. 163–175, 2010.

[21] K. Pruhs, "Competitive online scheduling for server systems," *SIGMETRICS Performance Evaluation Review*, vol. 34, no. 4, pp. 52–58, 2007.

[22] K. Kogan, A. López-Ortiz, G. Scalosub, and M. Segal, "Large profits or fast gains: A dilemma in maximizing throughput with applications to network processors," 2012, [Online] http://arxiv.org/abs/1202.5755.

[23] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Communications of the ACM*, vol. 28, no. 2, pp. 202–208, 1985.

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

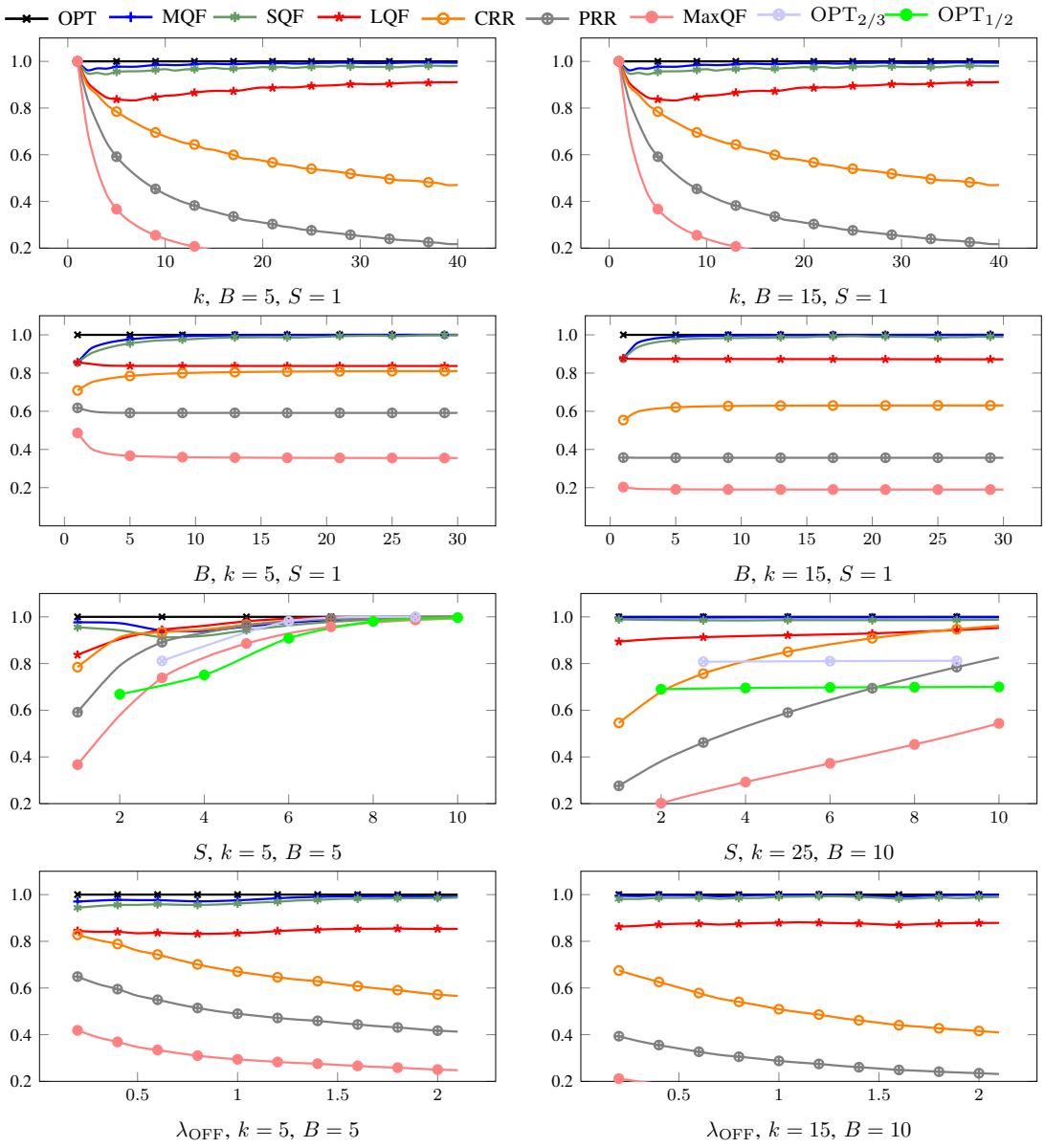

Fig. 3. Simulation study; y-axis, competitiveness; x-axis, top to bottom: max required processing k, buffer size B, speedup S, intensity λ_{OFF}.