Queueing in the mist: Buffering and scheduling with limited knowledge

Queueing in the Mist: Buffering and Scheduling with Limited Knowledge

Itamar Cohen and Gabriel Scalosub
Department of Communication Systems Engineering, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel
Email: itamarq@post.bgu.ac.il, sgabriel@bgu.ac.il

***Abstract*—Scheduling and managing queues with bounded buffers are among the most fundamental problems in computer networking. Traditionally, it is often assumed that all the properties of each packet are known immediately upon arrival. However, as traffic becomes increasingly heterogeneous and complex, such assumptions are in many cases invalid. In particular, in various scenarios information about packet characteristics becomes available only after the packet has undergone some initial processing.**

In this work, we study the problem of managing queues with limited knowledge. We start by showing lower bounds on the competitive ratio of any algorithm in such settings. Next, we use the insight obtained from these bounds to identify several algorithmic concepts appropriate for the problem, and use these guidelines to design a concrete algorithmic framework. We analyze the performance of our proposed algorithm, and further show how it can be implemented in various settings, which differ by the type and nature of the unknown information. We further validate our results and algorithmic approach by a simulation study that provides further insights as to our algorithmic design principles in face of limited knowledge.

I. Introduction

Some of the most basic tasks in computer networks involve scheduling and managing queues equipped with finite buffers, where the primary goal in such settings is maximizing the throughput of the system. The always-increasing heterogeneity and complexity of network traffic makes the challenge of maximizing the throughput ever harder, as the packet processing required in such queues span a plethora of tasks including various forms of DPI, MPLS and VLAN tagging, encryption / decryption, compression / decompression, and more.

The most prevalent assumption in the research studying these problems is that the various properties of any packet – e.g., its QoS characteristic, its required processing, its deadline – are known upon its arrival. However, this assumption is in many cases unrealistic. For instance, when a packet is recursively encapsulated a few times by MPLS, GRE or IPSec, it is hard to determine in advance the total number of processing cycles that such a packet would require. Furthermore, the QoS features of a packet are commonly determined by its flow ID, which is in many cases known only after parsing [1].

In data center networks and in software defined networks, a switch first looks for the forwarding and priority information in a local cache [2], [3]. A cache miss, which is unpredictable by nature, results in forwarding of the packet to the switch software or to a central controller, thus requiring a few additional processing cycles before the packet can be transmitted.

However, packet's characteristics usually become known once some initial processing is performed. This is common in many of the applications just described. Furthermore, for traffic corresponding to the same flow, it is common for characteristics to be unknown when the first few packets of the flow arrive at a network element, and once these properties are unraveled, they become known for all subsequent packets of this flow.

In this work we address such scenarios where the characteristics of some arriving traffic are unknown upon arrival, and are only revealed when a packet has undergone some initial processing (parsing), "causing the mist to clear". We model and formulate the problem of maximizing the profit obtained from delivered packets in such settings. We further show lower bounds on the competitive ratio of any randomized algorithm for the problem, and devise online algorithms with proven analytic guarantees on their expected performance. Lastly, we validate and evaluate the performance of our proposed solutions via a simulation study which sheds further light on the performance of our algorithms, beyond that provided by our analysis. We believe that our algorithmic design concepts might be applicable to additional scenarios as well.

Due to space constraints, most proofs are omitted and can be found in [4].

A. System Model

Our system model consists of the following modules: (a) a finite input buffer, which can contain at most B packets, (b) a buffer manager, which performs admission control, (c) a scheduler, which decides which of the pending packets should be processed, and (d) a processing element (PE), which processes the scheduled packet.

We divide time into discrete cycles. Each cycle consists of three steps: (i) *Transmission*, where fully-processed packets leave the queue, (ii) *Arrival*, where new packets may arrive, and the buffer manager decides which of them should be retained in the queue, and which of the currently buffered packets should be pushed-out and dropped, and (iii) *Processing*, where the scheduler assigns a single packet for processing by the PE.

We consider unit-size packets arriving at the queue. Upon its arrival, the characteristics of each packet may be *known* (resp. *unknown*), in which case we refer to the packet as a K*-packet* (resp. U*-packet*). We let M denote the maximum number of U-packets that may arrive in any single cycle. We assume that upon processing a U-packet for the first time, its properties become known [5].

Each arriving packet p has some (1) required number of processing cycles (*work*), $w(p) \in \{1, ..., W\}$, and (2) *profit* $v(p) \in \{1, ..., V\}$. We use the notation (w, v)*-packet* to denote a packet with work w and profit v.

The *head-of-line* (HoL) packet at time t (for a given algorithm Alg) is the highest priority packet stored in the buffer just prior to the processing step of cycle t, namely, the packet to be scheduled for processing in the processing step of t. We say the buffer is *empty* at cycle t if there are no packets in the buffer after the transmission step of cycle t.

We focus our attention on algorithms which are responsible for both managing the buffer and scheduling the packets for processing. In particular, we focus on algorithms targeted at maximizing the *throughput* of the queue, i.e. the overall profit from all packets successfully transmitted out of the queue.

We evaluate the performance of online algorithms using competitive analysis [6], [7]. An algorithm Alg is said to be c-competitive if for every finite input sequence σ, the throughput of *any* algorithm for this sequence is at most c times the throughput of Alg ($c \geq 1$). We let OPT denote any (possibly clairvoyant) algorithm attaining optimal throughput.

B. Related Work

Competitive algorithms for scheduling and management of bounded buffers have been extensively studied for the past two decades. An extensive survey of these models and their analysis can be found in [8]. While traditionally the research assumed uniform work, some recent studies addressed the problem of heterogeneous work, combined with either homogenous profits [9] or heterogeneous profits [10]. In particular, [10] showed that the competitive ratio of some straight-forward deterministic algorithms for the problem of heterogeneous work combined with heterogeneous profits is linear in either the maximal work W, or in the maximal profit V, even when the characteristics of all packets are known upon arrival. These results motivate our focus on randomized algorithms. These problems are also related to job scheduling in multi-threaded environments [11].

While most of the literature above assumed that all the characteristics of packets are known upon arrival, this assumption was put in question recently [5] by noting that it is often invalid. However, the main problem addressed in [5] revolved around developing schemes for transmitting packets of the same flow in-order, even when their required processing times are unknown upon arrival.

Maybe closest to our work is the recent work considering serving in the dark [12], which investigates an extreme case, where the online algorithm learns the profit from a packet only after transmitting it. This work considers highly oblivious algorithms, whereas our model and our proposed algorithms dwell in a middle-ground between the well studied models with complete information, and these recent oblivious settings. Our work further considers traffic with variable processing requirements, whereas [12] focuses on settings where all packets require only a single processing cycle, and they differ only by their profit.

II. Algorithmic Concepts

In this section we describe the algorithmic concepts underlying our proposed algorithms for dealing with scenarios of limited knowledge.

Random selection: Ideally, we would like every arriving U-packet to have at least some minimal probability of being accepted and parsed, thus avoiding a scenario where OPT successfully transmits a bulk of "good" packets which the online algorithm discards. An intuitive way to do that is to pick the unknown packets at random.

Speculatively Admit: Competitive algorithms must ensure they retain throughput from both K-packets and U-packets. Furthermore, once a U-packet is accepted, there is a high motivation to reveal its characteristics as soon as possible, thus making educated decisions in the next cycles. We therefore propose to *speculatively* over-prioritize unknown packets over known packets in certain cycles. The act of making such a choice in some cycle t is referred to as *admitting*, in which case cycle t is referred to as an *admittance cycle*. A U-packet retained due to such a choice is referred to as an *admitted packet*.

Randomly classify and select: Intuitively, as unknown packet characteristics are drawn from a wider range of values, the task of maximizing throughput becomes harder, especially when compared to the optimal throughput possible. To deal with this diversity, we apply a classify and select scheme [13], which enables us to provide analytic guarantees on the expected performance of our algorithms.

Alternate between fill & flush: This paradigm is especially crucial in cases of limited information. The main motivation for this approach is that whenever a "good" buffer state is identified, the algorithm should focus all its efforts on monetizing the current state, maybe even at the cost of dropping packets indistinctly.

III. Competitive Algorithms

In this section we study competitive online algorithms for the problem of buffer management and scheduling with limited knowledge. We first show a lower bound on the competitive ratio of every online random algorithm for this problem, and later present a competitive online algorithm, and provide a rigorous analysis of its performance.

The following theorem shows a lower bound on the competitive ratio attainable by any randomized algorithm for our problem. We note that this bound essentially relates any algorithm's performance with the amount of uncertainty and heterogeneity in the underlying traffic (proof omitted).

Theorem 1. *The competitive ratio of any randomized algorithm is* $\Omega(\min\{VW, M\})$.

In what follows we present a basic competitive online algorithm for the problem of buffering and scheduling with limited knowledge. We later describe in Section IV several improved variants of this algorithm.

For simplicity of analysis and algorithm presentation, we assume that the values of W and V are known to the algorithm

in advance. However, it is possible to remove this assumption without harming the performance of our algorithm (proof omitted). We further note that neither of our proposed solutions require knowing the value of M in advance.

A. High-level Description of Proposed Algorithm

Our algorithm is designed according to the algorithmic concepts presented in Section II as follows.

Randomly select and speculatively admit: In every cycle t during which a U-packet arrives, the algorithm picks t as an *admittance cycle* with some probability r (to be determined in the sequel). In every admittance cycle the algorithm picks a single U-packets arriving at t to serve as the *admitted* packet. This U-packet is chosen uniformly at random out of all U-packets arriving at t. At the end of the arrival step, the algorithm schedules the admitted U-packet (if one exists) for processing, hence *parsing* the packet. If no such U-packet exists, or if t is not an admittance cycle, then the Head-of-Line (HOL) packet is scheduled for processing. The exact determination of the HoL packet will be detailed later.

Randomly classify and select: We implicitly partition the possible types of arriving packets into classes $C_1, C_2, \ldots C_m$; the criteria for partitioning and the exact value of m will be specified later. Our algorithm picks a single *selected* class, uniformly at random from the m classes. Our goal is to provide *guarantees* on the performance of our proposed algorithm for packets belonging to the selected class, which is henceforth denoted G. Packets which belong to the selected class are referred to as *G-packets*. Following our previously introduced notation, known (unknown) packets that belong to the selected class, i.e., G-packets for which their attributes are known (unknown), are denoted as *G^K-packets* (*G^U-packets*).

Focusing solely on packets belonging to G may seem like a questionable choice, especially if there are few packets arriving which belong to this class, or if the characteristics of packets belonging to this class are poor. However, this naive description is meant only to simplify the analysis. In Section IV we show how to remedy this naive approach, while keeping the analytic guarantees intact.

Alternate between fill & flush: Our algorithm will be alternating between two states: the *fill* state, and the *flush* state. We define an algorithm to be *Hfull* if its buffer is filled with known G-packets. Once becoming Hfull, our algorithm switches to the flush state, during which it discards all arriving packets and continuously processes queued packets. Once the buffer empties, the algorithm returns to the fill phase. Again, in Section IV we show how to remedy this naive approach.

B. The Randomly Classify and Select Mechanism

We now turn to define the various classes considered by our algorithm. We say a packet p with $w(p) > 1$ is of *work-class* $C_i^{(W)}$ if $\lceil \log_2 w(p) \rceil = i$. If $w(p) = 1$ we assign it to work class $C_1^{(W)}$. Similarly, we say p with $v(p) > 1$ is of *profit-class* $C_j^{(P)}$ if $\lceil \log_2 v(p) \rceil = j$, and we assign it to profit class $C_1^{(P)}$ if $v(p) = 1$. Note that the work-class of a packet p is defined statically by the total work of p, and does not depend upon its remaining processing cycles, which may change over time.

This yields a collection of $\log_2 W$ work-classes, and $\log_2 V$ profit-classes. Lastly, we say a packet p is of *combined-class* $C_{(i,j)}$ if it is of work-class $C_i^{(W)}$ and of profit-class $C_j^{(P)}$. Upon initialization, the algorithm chooses the selected combined-class $G = C_{(i^*,j^*)}$ by picking $i^* \in \{1, \ldots, \log_2 W\}$ and $j^* \in \{1, \ldots, \log_2 V\}$, each chosen uniformly at random.

C. The SA Algorithm

We now describe the specifics of our algorithm, Speculatively Admit (SA), and analyze its performance. The pseudo-code of SA, depicted in Algorithm 1, uses the following procedures:

- *DecideAdmittance()* returns true with probability r.
- *UpdatePhase()*: if the buffer is empty (rsep., Hfull), set *phase* to fill (resp., flush). Otherwise, *phase* is unchanged.
- *Admit(p)*: If at cycle t $admittance$ is true and p is a U-packet, then admit p w.p. $1/N_t$, where N_t is the number of U-packets that have arrived in cycle t by the arrival of p (including p itself). This procedure essentially performs reservoir sampling [14].
- *SortQueue()* sort queued packets in G^K-first order, breaking ties by FIFO

Once in the arrival step, the algorithm updates its phase (line 1). If the phase is flush, the algorithm skips the while loop (lines 3-12), thus discarding all arriving packets. If the phase is fill, the algorithm greedily accepts every arriving packet as long as its buffer is not full (lines 4-5). If the buffer is full, however, the algorithm accepts an arriving packet only if it is either a G^K-packet, or an admitted U-packet (lines 6-8). In either of these cases, the last packet in the queue is dropped (line 7), so as to free space for the accepted packet. While

Algorithm 1 SA: at every time slot t after transmission

Arrival Step:

```
1:  phase = UpdatePhase()
2:  admittance = DecideAdmittance()
3:  while phase == fill and exists arriving packet p do
4:      if buffer is not full then
5:          accept p
6:      else if p is a G^K-packet or Admit(p) then
7:          drop packet from tail
8:          accept p
9:      end if
10:     phase = UpdatePhase()
11:     SortQueue()
12: end while
```

Processing Step:

```
13: if phase == fill and there exists an admitted packet p then
14:     move p to the HoL
15: end if
16: process HoL-packet
17: phase = UpdatePhase()
18: SortQueue()
```

in the processing step, if the algorithm is in the fill phase and there exists an admitted packet, the algorithm pushes it to the HoL, so as to parse it and reveal its characteristics (lines 13-15). Finally, the algorithm updates its phase and sorts the queued packets in G^K-first order each time it either accepts or processes a packet (lines 10-11 and 17-18).

We now turn to show an upper bound on the performance of our algorithm (for $W, V > 1$).

Theorem 2. SA *is* $O(\frac{M}{r} \log_2 W \log_2 V)$*-competitive.*

Proof sketch: We first prove the following propositions: (a) The algorithm never drops a G^K-packet. (b) In every admittance cycle t, SA's admitted packet is chosen uniformly at random out of all U-packets arriving at t.

As a result of the two propositions above, the overall number of G-packets transmitted by SA is at least an $\frac{r}{M}$ fraction of the G-packets accepted by an optimal policy during a fill phase.

We then use the fact that every class $C_{(i,j)}$ is the selected class with probability $O(\frac{1}{\log_2 W \cdot \log_2 V})$ to show that the expected performance of SA is at least an $O(\frac{M}{r} \log_2 W \log_2 V)$ fraction of the best performance possible. □

Our analysis shows that the best bound on the competitive ratio is attained for $r = 1$, i.e., every cycle where we have U-packets arriving should be an admittance cycle. In practical scenarios, however, one might want to be more conservative in choosing admittance cycles. E.g., one might choose $r < 1$ so as to allow non-parsing cycles even when U-packets arrive.

We note that when a characteristic consists of a small set of potential values, the logarithmic dependency on the *maximal value* of the characteristic can be transformed to a linear dependency on the *number of distinct values* for this characteristic. Furthermore, it is possible to implement SA even when the values of W and V are not known in advance, without any performance degradation (proofs omitted).

IV. Improved Algorithms

Algorithm SA selects a single class uniformly at random so that the characteristics of packets on which it focuses differ by at most a constant factor. This gives the sense of "uniformity" of traffic, which in turn reduces the variability of characteristics of packets on which the algorithm focuses. However, in practice there are various cases where the strict decisions made by SA can be relaxed without harming its competitive performance guarantees. In practice, such relaxations actually allow obtaining a throughput far superior to that of SA. In what follows we describe such modifications, which we incorporate into our improved algorithm, SA*. We note that all our performance guarantees for SA still hold for SA* (proofs omitted).

Class closure: Given any partitioning of packets into classes as described in Section III-B, we let the (i,j)*-closure class* be defined as $C^*_{(i,j)} = \bigcup_{i' \le i, j' \ge j} C_{(i',j')}$. This definition effectively assigns any packet which is at least as good as any packet in $C_{(i,j)}$, to the (i,j)-closure class. We emphasize that any such packet p must satisfy both $w(p) \le 2^i$ and $v(p) \ge 2^{j-1}$. We let SA* denote the algorithm where the selected class G is chosen to be $C^*_{(i,j)}$, for some values of i, j chosen uniformly at random from the appropriate sets.

Fill during flush (pipelining): Algorithm SA was defined such that no arriving packets are ever accepted during the flush phase. In practice, however, accepting packets during a flush phase cannot harm the analysis, nor the actual performance, if this is done prudently: packets which arrive during the flush phase are accepted according to the same priority suggested by the algorithm's behavior in the fill phase. Furthermore, packets which arrive during the flush phase are stored in the buffer, but never scheduled for processing before all B packets that are stored in the buffer when it turns Hfull are transmitted.

Improved scheduling: SA sorts the queued packets in G^K-first order. For simplicity of presentation, we assumed in Section III that within the set of G^K-packets, as well as within the set of non-G^K-packets, packets are internally ordered by FIFO. However, one may consider other approaches as well to performing such scheduling for each of these sets, while maintaining G^K-first order between the sets. In Section V we suggest different scheduling regimes, and study their performance. We emphasize that the packet scheduled for processing during an admittance cycle remains a U-packet, which is selected uniformly at random from the arriving U-packets at this cycle.

V. Simulation Study

In this section we present the results of our simulation study intended to validate our theoretical results, and provide further insight into our algorithmic design.

A. Simulation Settings

We simulate a single queue in a gateway router which handles a bursty arrival sequence of packets with high work requirements (corresponding, e.g., to IPSec packets, requiring AES encryption/decryption) as well as packets with low work requirements (such as simple IP packets requiring merely IPv4-trie processing). Arriving packets also have arbitrary profits, modeling various QoS levels.

Our traffic is generated by a Markov modulated Poisson process (MMPP) with two states, LOW and HIGH, such that the HIGH (resp., LOW) state generates an average of 10 (resp., 0.5) packets per cycle. The average duration of LOW-state periods is W times longer than the average duration of HIGH-state periods, so as to potentially allow some traffic arriving during the HIGH-state to be drained during the LOW-state.

We do not deterministically bound the maximum number, M, of U-packets arriving in a cycle, but rather control the expected intensity of U-packets by letting each arriving packet be a U-packet with some probability $\alpha \in [0,1]$. The expected number of U-packets per cycle during the HIGH state is therefore 10α.

In real-life scenarios, the maximum work, W, required by a packet, is highly implementation-depended. It depends on the specific hardware, PEs, and software modules. However, some studies indicate that W is two orders of magnitude larger than the work required for doing a fundamental work ("parsing"), such as a IPv4-Trie search or a classification of a packet [15].

We therefore set the maximum work required by a packet to $W = 256$ throughout this section. The maximum profit, V, associated with a packet, depends both on implementation details, as well as on proprietary commercial and business considerations. In order to have a diverse set of values, which model distinct QoS requirements, we set the maximum profit associated with a packet to $V = 16$.

The values $W = 256$ and $V = 16$ imply a total of $8 \cdot 4 = 32$ potential classes for the algorithm to select from. The value of each characteristic for each packet is drawn from a Pareto-distribution, with average and standard deviations of 17.97 and 22.22 for packet work, and 3.66 and 3.20 for packet profit.

We assume that $B = 10$, $r = 1$ and each arriving packet is a U-packet with probability $\alpha = 0.3$. We thus obtain that the expected number of U-packets arriving during the HIGH state is $0.3 \cdot 10 = 3$ per cycle.

As a benchmark which serves as an upper bound on the optimal performance possible, we consider a relaxation of the offline problem as a knapsack problem. Arriving packets are viewed as items, each with its size and value (corresponding to the packet's work and profit, resp.) The allocated knapsack size equals the number of time slots during which packets arrive. The goal is to choose a highest-value subset of items which fits within the given knapsack size. This is indeed a relaxation of the problem of maximizing throughput during the arrival sequence in the offline setting, since the knapsack problem is not restricted by any finite buffer size during the arrival sequence, nor by the arrival time of packets (e.g., it may "pack" packets even before they arrive).

We approximate an upper bound on the performance of OPT by employing the classic 2-approximation greedy algorithm for solving the knapsack problem [16]. To allow gain from packets which reside in its buffer at the end of the arrival sequence, we simply allow the offline approximation an additional throughput of BV for free, which is an upper bound on the benefit it may achieve after the arrival sequence ends.

We compare the performance of studied algorithms by evaluating their *performance ratio*, which is the ratio between the algorithm's performance and that of our approximate upper bound on the performance of OPT. We compare the performance of the following algorithms:

1) *FIFO*: A simple greedy non-preemptive FIFO discipline that simply accepts packets and processes each packet until completion, regardless of its required work or value.
2) SA: Algorithm SA, described in Section III.
3) SA^* *FIFO*: Algorithm SA^* where packets are processed in FIFO order.
4) SA^* *W-Then-V*: Algorithm SA^* where packets are processed in increasing order of remaining work, breaking ties in decreasing order of profit.
5) SA^* *EFFECT*: Algorithm SA^* where the packets are processed in decreasing order of their profit-to-work ratio, commonly referred to as *effectiveness*.

For each scenario we show the average of running 100 independently-generated traces of 10K packets each. In all simulations the standard deviation was below 0.035.

B. Simulation Results

Figures 1 and 2 show the results of our simulation study. First we note that SA exhibits a very low performance ratio, similar to that of a simple FIFO (which disregards packets parameters altogether). This is due to the fact that SA focuses only on a specific class, which consists of a relatively small part of the input, and it thus spends processing cycles on packets that would not be eventually transmitted.

For the variants of SA^* we consider, in all simulations the best performance is achieved by SA^* EFFECT, followed by SA^* W-THEN-V. FIFO scheduling, in spite of it being simple and attractive, comes in last in all scenarios. This behavior is explained by the fact that both former scheduling policies in SA^* clear the buffer more effectively once it is Hfull. The latter FIFO scheduling approach clears the buffer in an oblivious manner, and therefore doesn't free up space for new arrivals fast enough. We now turn to discuss each of the scenarios considered in our study.

1) The Effect of Selected Class: These results shed light on the effect of the class selected by an algorithm on its performance. Figure 1 shows the results where the selected profit-class is 1, which makes SA^* allow all profits, and the choice of work-class i^* varies. The most interesting phenomena is exhibited by SA^* FIFO. Its performance is very poor if the work-class may contain packets requiring very little work. This is due to the fact that only a small fraction of the traffic requires this little work, and the algorithm scarcely arrives at being Hfull. As a consequence, the algorithm handles many low-priority packets, which are handled in FIFO order, giving rise to far-from-optimal decisions. The algorithm steadily improves up to some point, and then its performance deteriorates fast as it assigns high-priority to packets with increasingly higher processing requirements. In this case the algorithm becomes Hfull too frequently, and allocates many processing cycles to low-effectiveness packets. The maximum performance is achieved for $i^* = 3$, which implies that the algorithm flushes whenever its buffer is filled up with packets whose work is at most $2^{i^*} = 8$. This value suffices to allow the algorithm to prioritize a rather large portion of the arrivals (recalling the Pareto distribution governing packet work-values), while ensuring the processing toll of high-priority packet is not too large. This strikes a (somewhat static) balance between the amount of work required by a packet, and its expected potential profit. The other variants of SA^* exhibit a gradually decreasing performance, due to their higher readiness to compromise over the required work of packets they deem as high-priority traffic. SA shows a similar performance deterioration, for a similar reason, when the selected work-class i^* is increased from 1 up to 6. However, when increasing i^* above 6, SA's performance increases again. This improvement is explained by the fact that, due to the Pareto-distribution of the work values, the number of packets which belong to each work-class rapidly diminishes when switching to work-class indices closest to the maximum of 8. In such a case, SA is coerced to process also packets which do not belong to the selected class – namely,

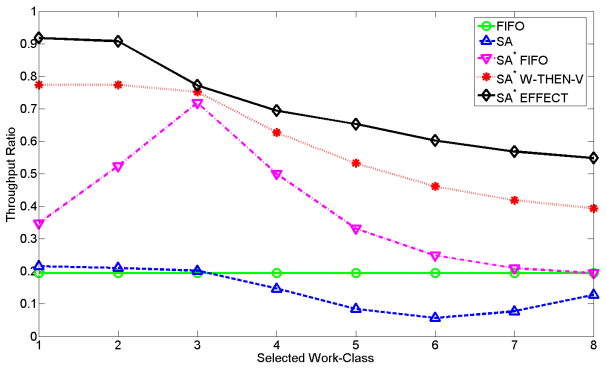

Fig. 1: Effect of chosen work-class i^*

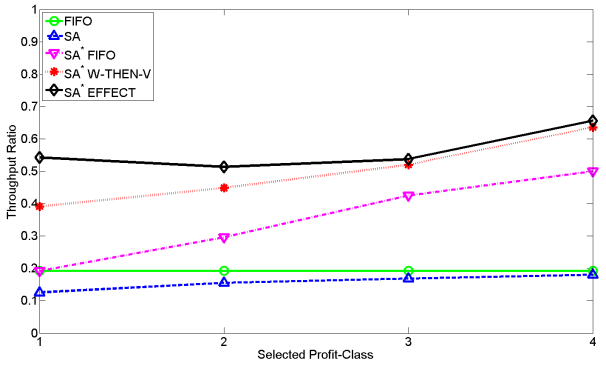

Fig. 2: Effect of chosen profit-class j^*

packets with *lower* work – which somewhat compensates for the poor choice of the work-class. We verified this explanation by additional simulations (not shown here), in which the work-class of packets was chosen from the uniform distribution. In such a case, where there is an abundance of packets from every possible work-class, the performance of SA consistently degrades with the increase of i^*, which implies a poorer choice of work-class.

Similar phenomena are exhibited in Figure 2, where we consider the effect of the profit-class j^* selected by an algorithm on its performance. In this set of simulations all work-values were allowed (i.e., the selected work-class is 8). In this scenario the performance of all algorithms improves as the selected profit-class index increases, and the algorithms are able to better restrict their focus on high profit packets as the packets receiving high-priority. We note the fact that SA* FIFO and regular FIFO have a matching performance in the case the selected profit-class is 1, since in this case SA* FIFO is identical to plain FIFO (since it simply indiscriminately accepts all incoming packets in FIFO order).

In additional simulations (omitted due to space constraints) we studied the effect of the number of U-packets per cycle; and of the intensity of exploring unknown packets. These simulations show that the performance of our proposed solutions *degrades* as the amount of uncertainty increases; and *increases* as we increase r, governing our exploration intensity. These results coincide with our analytic results, which further validate our algorithmic approach.

VI. Conclusions and Future Work

We consider the problem of managing buffers where traffic has unknown characteristics, namely required processing and profits. We devise algorithms for the problem, and show upper bounds on their competitive ratio. A simulation study then provides further insight as to their performance. Our work gives rise to a multitude of open questions, including: (i) closing the gap between our lower and upper bound for the problem, (ii) applying our proposed approaches to other limited knowledge networking environments, and (iii) devising additional algorithmic paradigms for handling limited knowledge in heterogeneous settings.

Acknowledgements

This research was supported by the Israel Science Foundation (grant No. 1036/14), the Research & Innovation action MIKELANGELO (project no. 645402) co-funded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme, and the Neptune Consortium, administered by the Israeli Ministry of Economy and Industry.

References

[1] C. Kozanitis, J. Huber, S. Singh, and G. Varghese, "Leaping multiple headers in a single bound: wire-speed parsing using the kangaroo system," in *INFOCOM*, 2010, pp. 830–838.

[2] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *ACM SIGCOMM Computer Communication Review*, vol. 39, 2009, pp. 39–50.

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *IEEE/ACM Transactions on Networking (TON)*, vol. 17, no. 4, pp. 1270–1283, 2009.

[4] [Online]. Available: http://tinyurl.com/lj3l4vs

[5] A. Shpiner, I. Keslassy, and R. Cohen, "Scaling multi-core network processors without the reordering bottleneck," in *HPSR*, 2014, pp. 146–153.

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

[7] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*. cambridge university press, 2005.

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

[9] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal, "Providing performance guarantees in multipass network processors," *IEEE/ACM Transactions on Networking (TON)*, vol. 20, no. 6, pp. 1895–1909, 2012.

[10] P. Chuprikov, S. Nikolenko, and K. Kogan, "Priority queueing with multiple packet characteristics," in *INFOCOM*, 2015, pp. 1418–1426.

[11] K. Pruhs, "Competitive online scheduling for server systems," *ACM SIGMETRICS Perf. Eval. Review*, vol. 34, no. 4, pp. 52–58, 2007.

[12] Y. Azar and I. R. Cohen, "Serving in the dark should be done non-uniformly," in *ICALP*, 2015, pp. 91–102.

[13] B. Awerbuch, Y. Bartal, A. Fiat, and A. Rosén, "Competitive non-preemptive call control." in *SODA*, 1994, pp. 312–320.

[14] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[15] M. E. Salehi, S. M. Fakhraie, and A. Yazdanbakhsh, "Instruction set architectural guidelines for embedded packet-processing engines," *Journal of Systems Architecture*, vol. 58, no. 3, pp. 112–125, 2012.

[16] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. Cambridge University Press, 2011.