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. The techniques used in our proofs, which make use of a carefully crafted Markov process, may be of independent interest, and can potentially be used in other similar settings as well. 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 an extensive simulation study that provides further insights as to our algorithmic design principles in face of limited knowledge.

Keywords: Buffer management, queueing, scheduling, uncertainty, limited knowledge, competitive analysis, online algorithms

1. 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

An earlier version of this work was published in [1]. This work adds full proofs of all theorems, stronger lower bounds, an improved competitive algorithm, and an extended simulation study.

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 spans 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, PBB, 802.1Q, GRE or IPSec, it is hard to determine in advance the total number of processing cycles that such a packet would require [2, 3]. Furthermore, the QoS features of a packet are commonly determined by its flow ID, which is in many cases known only after parsing [3].

In data center network architectures such as PortLand [4], ingress switches query a cache for an application-to-location address resolution. A cache miss, which is unpredictable by nature, results in forwarding of the packet to the switch software or to a central controller, which performs a few additional processing cycles before the packet can be transmitted. Similarly, in the realm of Software Defined Networks, ingress switches query a cache for obtaining rules for a packet [5], which may also depend on priorities [6]. In such a case, a cache miss results in additional processing until the rules are retrieved and the profit from the packet is known.

In spite of this increased heterogeneity, and the fact that the processing requirement of a packet might not be known in advance, these characteristics usually become known once some initial processing is performed. This behavior 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. It therefore follows that only part of the arriving packets has unknown characteristics upon arrival, which become known after parsing.

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 analyze the performance of algorithms in such settings, and in particular we develop online scheduling and buffer management algorithms for the problem of maximizing the profit obtained from delivered packets, and provide guarantees on their expected performance using competitive analysis.

We focus on the general case of heterogeneous processing requirements (work) and heterogeneous profits [7]. We assume priority queueing, where the exact priorities depend on the specifics of the model studied. We present both algorithms and lower bounds for the problem of dealing with unknown characteristics in these models. Furthermore, we highlight some design concepts for settings where algorithms have limited knowledge, which we believe might be applicable to additional scenarios as well.

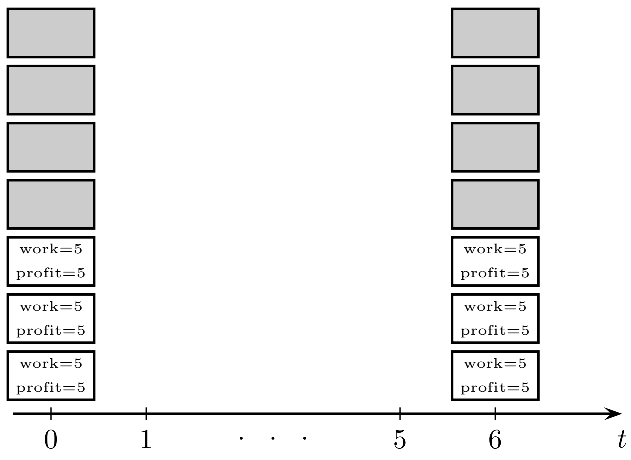


Figure 1: An illustrative example of an arrival sequence with known and unknown packets

As an illustration of the problem, assume we have a 3-slots buffer, equipped with a single processor, and consider the arrival sequence depicted in Fig. 1. In the first cycle we have seven unit-size packets arriving, out of which three will provide a profit of 5 upon successful delivery, each requiring 5 processing cycles (work). The characteristics of these three packets are known immediately upon arrival. The characteristics of the remaining four packets (marked gray) are *unknown* upon arrival. We therefore dub such packets U-packets (i.e., unknown packets). Each of these four U-packets may turn out to be either a "best" packet, requiring minimal work and having maximal profit; a "worst" packet, requiring maximal work and having minimal profit; or anything in between. Thus, already at the very beginning of this simple scenario, any buffering algorithm would encounter an *admission control* dilemma: how many U-packets to accept, if any? This dilemma can be addressed by various approaches including, e.g., allocating some buffer space for U-packets, accepting U-packets only when current known packets in the buffer are of poor characteristics, in terms of profit, or of profit to work ratio, etc. In case that the algorithm accepts U-packets, an additional question arises: which of the U-packets to accept into the buffer? Obviously, for any online deterministic algorithm there exists a simple adversarial scenario, which would cause it to accept only the "worst" U-packets (namely, packets with maximal work and minimum profit), while an optimal offline algorithm would accept the best packets. This motivates our decision to focus our attention on randomized algorithms.

We now turn to consider another aspect of handling traffic with some unknown characteristics. Assume the scenario continues with 5 cycles without any arrival, and then a cycle with an identical arrival pattern - namely, three known packets with both work and profit of 5 per packet, and four U-packets.

This sheds light on a *scheduling* dilemma: which of the accepted packets should better be processed first? every scheduling policy impacts the buffer space available in the next burst. For instance, a run-to-completion attitude would enable finishing the processing of one known packet by the next burst, thus allowing space for accepting a new packet without preemption. However, one may consider an opposite attitude - namely, parsing as many U-packets as possible, thus "causing the mist to clear", allowing more educated decisions, once there are new arrivals. In terms of priority queuing, this means over-prioritizing some U-packets, and allowing them to be parsed immediately upon arrival. We further develop appropriate algorithmic concepts based on the insights from this illustrative example in Section 3.

1.1. System Model

Our system model consists of four main modules, namely, (a) an input queue equipped with a finite buffer, (b) a buffer management module which performs admission control (c) a scheduler module which decides which of the pending packets should be processed, and (d) a processing element (PE), which performs the processing of a packet.

We divide time into discrete cycles, where each cycle represents a fixed time slot, and consists of three steps: (i) The *transmission* step, in which fully-processed packets leave the queue, (ii) the *arrival* step, in which new packets may arrive, and the buffer management module decides which of them should be retained in the queue, and which of the currently buffered packets should be pushed-out and dropped, and finally (iii) the *processing* step, in which the scheduler assigns a single packet for processing by the PE, which in turn processes the packet.

We consider a sequence of unit-size packets arriving at the queue. Upon its arrival, the characteristic of each packet may be *known* - in which case we refer to the packet as a K*-packet* (i.e., known packet); or *unknown* - in which case we refer to the packet as a U*-packet* (i.e., unknown packets). We let M denote the maximum number of U-packets that may arrive in any single cycle. We focus our attention on the case where $M > 0$, unless specifically stated otherwise.

Each arriving packet p has some (1) intrinsic benefit (*profit*) $v(p) \in \{1, \ldots, V\}$, and (2) required number of processing cycles (*work*), $w(p) \in \{W_0, W_0 + 1, \ldots, W\}$. Unless explicitly stated otherwise, we consider the most general case, namely, $W_0 = 1$. To simplify the expressions throughout the paper, we assume that both V and W are powers of 2.[1] We use the notation (w, v)*-packet* to denote a packet with work w and profit v. We note that the *uniform* case where all packets require the same amount of work, and all packets have the same profit, is trivial, since the simple run-to-completion policy is optimal. We therefore focus our attention on non-uniform traffic.

In our model, similarly to [8], upon processing a U-packet for the first time, its properties become known. We therefore refer to such a first processing cycle

[1] Our results degrade by a mere constant factor otherwise.

of a U-packet as a *parsing cycle*. Non-parsing cycles where the processor is not idle are referred to as *work cycles*.

The queue buffer can contain at most B packets. We assume $B \geq 2$, since the case where $B = 1$ is degenerate. 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 study *queue management* algorithms, which are responsible for both the buffer management and the scheduling of packets for processing. In particular, we focus our attention on algorithms targeted at maximizing the *throughput* of the queue, i.e. the overall profit from all packets successfully transmitted out of the queue. The throughput of algorithm Alg is denoted by $TP(\text{Alg})$. We use the terms throughput and performance interchangeably.

We evaluate the performance of online algorithms using competitive analysis [9, 10]. 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. An algorithm is said to be *greedy* if it accepts packets as long as there is available buffer space. We further focus our attention on *work-conserving* algorithms, i.e., algorithms which never leave the PE idle unnecessarily.

1.2. Related Work

Competitive algorithms for scheduling and management of bounded buffers have been extensively studied for the past two decades. The problem was first introduced in the context of differentiated services, where packets have uniform size and processing requirements, but some of the packets have higher priorities, represented by a higher profit associated with them [11, 12, 13]. The numerous variants of this problem include models where packets have deadlines or maximum lifetime in the switch [12], environments involving multi-queues [14, 15, 16, 17] and cases with packets dependencies [18, 19], to name but a few. An extensive survey of these models and their analysis can be found in [20].

While traditionally it was assumed that packets have heterogeneous profits but uniform work (processing requirements), some recent work introduced the complementary problem, of uniform profits with heterogeneous work [21]. This work presented an optimal algorithm for the fundamental problem, as well as online algorithms and bounds on the competitive ratio for numerous variants. Subsequent research investigated related problems with heterogeneous work combined with heterogeneous packet sizes [22], or with heterogeneous profits [7, 23]. In particular, [7] 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.

While most of the literature above assumed that all the characteristics of packets are known upon arrival, this assumption was put in question recently [8] by noting that it is often invalid. However, the main problem addressed in [8] revolved around developing schemes for transmitting packets of the same flow in-order, while our work focuses on maximizing throughput with limited buffering resources, and designing both buffer management and scheduling policies targeted at this objective.

Maybe closest to our work are the recent studies considering serving in the dark [24, 25], which investigate an extreme case where the online algorithm learns the profit from a packet only after transmitting it. These studies consider 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 [24, 25] focus on settings where all packets require only a single processing cycle, and they differ only by their profit.

The problem of optimal buffering of packets with variable work is closely related to the problem of job scheduling in a multi-threaded processor, which was extensively studied. A comprehensive survey of online algorithms for this problem can be found in [26]. This body of work, however, differs significantly from our currently studied model. The major differences are that packet buffering has to deal with limited buffering capabilities, and is targeted at maximizing throughput. Processor job scheduling, however, usually has no strict buffering limitations, and is mostly concerned with minimizing the response time.

1.3. Our Contribution

We introduce the problem of buffering and scheduling which aims to maximize throughput where the characteristics of some of the packets are unknown upon arrival. We focus our attention on traffic where every packet has some required processing cycles, and some profit associated with successfully transmitting it. We make no assumption on the underlying process generating traffic, thus rendering our results globally applicable.

In Section 2 we present lower bounds on the performance of any randomized algorithm for the problem. Specifically, we show that no algorithm can have a competitive ratio better than $\Omega(\min\{WV, M\})$, even against an adversary which can accommodate merely 2 packets in its buffer, where W and V denote the maximum work and profit of a packet, respectively, and M represents the maximum number of unknown packets which may arrive in any single cycle. We also prove stronger lower bounds for the general settings using a novel technique, in which we bound the expected number of packets in the buffer of an optimal offline algorithm by means of a Markov process.

In Section 3 We describe several algorithmic concepts tailored for dealing with unknown characteristics in such systems. We follow by presenting an algorithm that applies our suggested algorithmic concepts in Section 4. For the most general case we prove our algorithm has a competitive ratio of $O(M \log V \log W)$. We further show how to improve this bound in several important special cases.

In Sections 5-6 we present some modifications and heuristics applicable to our algorithm that, while leaving the worst-case guarantees intact, are designed to improve performance compared to the baseline algorithmic design. The modified algorithm can cope with cases where neither the maximal amount of work and profit, nor the maximum number of unknown packets per cycle, are known in advance.

We further validate and evaluate the performance of our proposed algorithms in Section 7 via an extensive simulation study. Our results highlight the effect the various parameters have on the problem, well beyond the insights arising from our rigorous mathematical analysis.

We conclude in Section 8 with a discussion of our results, and also highlight several interesting open questions.

2. Lower Bounds

In this section we present lower bounds on the competitive ratio of any randomized algorithm for our problem. These lower bounds serve two main objectives: (i) They represent the best competitive ratio which one can hope to achieve; and (ii) the hard scenarios used in the proofs of these lower bounds highlight the challenges which any competitive online algorithm would have to tackle.

2.1. Highly-restricted adversaries

In this section we prove lower bounds on the competitive ratio of any online algorithm for our problem, compared to a highly-restricted adversary which uses a buffer which can only store a single packet. This restriction on the amount of buffer space available for the adversary enables us to better highlight the scaling laws of the problem, depending on the various parameters.

Theorem 1. *If $V \geq 1$, $M \geq 1$ and the work of each packet is $w(p) \in \{W_0, W_0 + 1, \ldots, W\}$ where $W \geq 2$, then the competitive ratio of any randomized algorithm for non-uniform traffic is at least*

$$\frac{V(W-1)}{2W_0}\left[1-\left(1-\frac{1}{V(W-1)+1-W_0}\right)^{MW_0}\right],$$

even against an optimal offline algorithm which has a buffer which can only store a single packet.

Proof. Since traffic is non uniform, we are guaranteed to have $V(W-1)+1-W_0 \neq 0$. We prove the theorem using Yao's method [27], where we define a carefully crafted distribution over arrival sequences, and show a lower bound on the ratio between the expected performance of an optimal clairvoyant algorithm for the problem, and the expected performance of any *deterministic* algorithm for the problem. We will show that the claim is true even if the optimal offline algorithm uses a buffer that can hold only a single packet. We define the following collection of arrival sequences, where each arrival sequence has two phases: a *Fill phase*, and a *Flush phase*. The Fill phase consists of *iterations* as follows. Each iteration begins with W_0 cycles without arrivals; and continues with W_0 cycles with M U-packets arriving per cycle, where each packet is a (W_0, V)-packet with probability p, and a $(W, 1)$-packet with probability $(1-p)$, for some constant p to be determined later. The total number of cycles during the fill phase is N, where N is a large integer, so we have $\frac{N}{2W_0}$ iterations. Once the fill phase ends, it is followed by the Flush phase, which consists of BW cycles without arrivals. We note that due to the random choices of packets being either (W_0, V)-packets or $(W, 1)$-packets, the above structure induces a distribution over a collection of possible arrival sequences.

To simplify our analysis, we define the SubOPT policy, which works as follows: Within the fill phase, during each iteration, SubOPT accepts at most one (W_0, V)-packet which has arrived during the iteration, if such a packet exists. This packet is the one considered *picked* by SubOPT in that iteration. Starting from the second iteration, during the first W_0 cycles of each iteration, SubOPT processes the packet it picked during the previous iteration (if such a packet exists), and transmits it. During the flush phase, SubOPT processes and finally transmits the packet it picked during the last iteration.

It should be noted that SubOPT is neither greedy, nor work conserving. Moreover, the expected throughput of SubOPT clearly serves as a lower bound on the expected optimal throughput possible.

We have $\frac{N}{2W_0}$ iterations, and the probability that SubOPT successfully picks a (W_0, V)-packet during an iteration is exactly the probability of there being a (W_0, V)-packet arriving during that iteration, which is $1-(1-p)^{MW_0}$. The throughput of SubOPT, which we recall is denoted by $TP(\text{SubOPT})$, therefore satisfies

$$TP(\text{SubOPT}) \geq \frac{NV}{2W_0}[1-(1-p)^{MW_0}] \tag{1}$$

We now turn to consider the expected performance of any deterministic algorithm Alg for the problem. We first assume that Alg begins the flush phase with a buffer full of (W_0, V)-packets, all of them unparsed. This provides Alg with a profit of BV during the flush phase, while still having N processing cycles during the fill phase for processing additional packets. This profit is clearly an upper bound on the maximum possible throughput attainable by Alg from packets transmitted during the flush phase, regardless of when they were processed. For evaluating the gain of Alg during the fill phase, it therefore suffices to consider only packets which Alg fully processes during this phase.

Consider now the profit of Alg from packets transmitted during the fill phase. Recall that we assume that Alg is work-conserving. We assume that Alg is also greedy, that is, Alg never discards a packet when its buffer is not full; being greedy cannot decrease Alg's performance. Alg has packets to process during the entire fill phase, except for the first W_0 cycles (where there are no arrivals yet), namely, for $N' = N - W_0$ cycles. Furthermore, since Alg is assumed to always accept packets when the buffer is not full, and is work conserving, there exists some $0 < r \leq 1$ such that the number of parsing, and work, cycles performed by Alg are $N'r$, and $N'(1-r)$, respectively.

Consider a case where Alg reveals a (W_0, V)-packet q. Then, processing q and finally transmitting it would surely not decrease the throughput of Alg when contrasted with the alternative of dropping q. Thus, the best deterministic algorithm Alg would work at least $W_0 - 1$ work cycles per each parsing cycle, in which a (W_0, V)-packet is parsed (recall that we are merely interested in packets, which Alg fully processes and transmits during the fill phase). Therefore, the total number of work cycles contributing to the transmission of such packets is at least $W_0 - 1$ times larger then the expected number of parsing cycles, in which a (W_0, V)-packet is revealed: $N'(1-r) \geq N'rp(W_0 - 1)$.

If the total number of work cycles during the fill phase exceeds the number of cycles which are necessary for transmitting all the parsed (W_0, V)-packets, Alg may work also on $(W, 1)$-packets. Namely, if $N'(1-r) > N'rp(W_0-1)$, then Alg may work on $(W, 1)$-packets for $N'(1-r) - N'rp(W_0 - 1)$ cycles, transmitting at most one $(W, 1)$-packet once in $W - 1$ such cycles.

Combining the above reasoning we conclude that the overall throughput of Alg satisfies

$$\begin{aligned} TP(\text{Alg}) &\leq N'rpV + \frac{N'(1-r) - N'rp(W_0-1)}{W-1} + BV \\ &= (N - W_0)\left[Vrp + \frac{(1-r) - rp(W_0-1)}{W-1}\right] + BV \end{aligned} \tag{2}$$

Considering the ratio between the lower bound on the expected performance of SubOPT (as captured by Eq. 1) and the upper bound on the expected performance of Alg (as captured by Eq. 2) and letting $N \to \infty$, we conclude that no algorithm can have a competitive ratio better than

$$\frac{V(W-1)}{2W_0} \cdot \frac{1 - (1-p)^{MW_0}}{Vrp(W-1) + 1 - r - rp(W_0 - 1)}$$

By choosing $p^* = [V(W-1) + 1 - W_0]^{-1}$, the result follows. $\square$

We now aim to relate the lower bound established in Theorem 1 to a simpler and more intuitive function of M, V and W. We do so by means of two propositions, which relate the bound to either $\Omega(M)$ or $\Omega(VW)$ for different ranges of M. In the propositions we use our notation $p^* = [V(W-1) + 1 - W_0]^{-1}$ from the proof of Theorem 1. Using this notation, note that Theorem 1 shows that the competitive ratio is at least

$$\frac{V(W-1)}{2W_0}\left[1 - (1-p^*)^{MW_0}\right].$$

In the proofs of both propositions we will repeatedly use the following simple inequality, which holds for any $W_0 \geq 1$:

$$\frac{1}{V(W-1)} = \frac{1}{\frac{1}{p^*} + W_0 - 1} \leq p^*. \tag{3}$$

The following proposition shows that if M is relatively small, then the lower bound established in Theorem 1 is $\Omega(M)$.

Proposition 2. *If* $V \geq 1, W_0 \geq 1, W \geq 2$ *and* $1 \leq M \leq \frac{V(W-1)}{W_0}$, *then*

$$\frac{V(W-1)}{2W_0}\left[1-(1-p^*)^{MW_0}\right] \geq \frac{M}{4}$$

Proof. We show by induction on n that for any $1 \leq n \leq V(W-1)$

$$(1-p^*)^n \leq 1 - \frac{n}{2V(W-1)}. \tag{4}$$

By setting $n = M \cdot W_0$, which is at most $V(W-1)$ by our assumption on M, and applying some algebraic manipulation, the result follows.

For $n = 1$, Eq. 4 reduces to requiring that $\frac{1}{2V(W-1)} \leq p^*$, which holds true due to Eq. 3. For the induction step, by the induction hypothesis on n we have

$$(1-p^*)^{n+1} \leq (1-p^*)\left[1-\frac{n}{2V(W-1)}\right].$$

It therefore suffices to prove that

$$(1-p^*)\left[1-\frac{n}{2V(W-1)}\right] \leq 1-\frac{n+1}{2V(W-1)},$$

which is equivalent to requiring that

$$\frac{1}{2V(W-1)} \leq p^*\left[1-\frac{n}{2V(W-1)}\right].$$

By Eq. 3 we have $\frac{1}{2V(W-1)} \leq \frac{p^*}{2}$, which implies that it suffices to show that

$$\frac{p^*}{2} \leq p^*\left[1-\frac{n}{2V(W-1)}\right]$$

which is satisfied for every $n \leq V(W-1)$. □

The following proposition shows that if M is relatively large, then the lower bound established in Theorem 1 is $\Omega\left(\frac{VW}{W_0}\right)$.

Proposition 3. *If* $V \geq 1, W_0 \geq 1, W \geq 2$ *and* $M > \frac{V(W-1)}{W_0}$, *then*

$$\frac{V(W-1)}{2W_0}\left[1-(1-p^*)^{MW_0}\right] > \frac{e-1}{4e}\cdot\frac{VW}{W_0}$$

Proof. By our assumption on M, and using Eq. 3, we have $M \cdot W_0 > V(W-1) \geq \frac{1}{p^*}$. It follows that $M \cdot W_0 = a\frac{1}{p^*}$ for some $a > 1$, which in turn implies that

$$(1-p^*)^{M \cdot W_0} = \left[(1-p^*)^{\frac{1}{p^*}}\right]^a \leq e^{-a} < e^{-1}.$$

It follows that

$$\begin{aligned}
\frac{V(W-1)}{2W_0}\left[1-(1-p^*)^{M \cdot W_0}\right] &\geq \frac{V(W-1)}{2W_0}\left(1-\frac{1}{e}\right) \\
&= \frac{VW}{2W_0} \cdot \frac{W-1}{W}\left(1-\frac{1}{e}\right) \\
&\geq \frac{e-1}{4e}\frac{VW}{W_0}
\end{aligned}$$

□

Assigning $W_0 = 1$ in Theorem 1 and Propositions 2 and 3 implies the following corollary:

Corollary 4. *The competitive ratio of any randomized algorithm is* $\Omega(\min\{VW, M\})$.

In the special case of uniform-profits, we are essentially interested in maximizing the overall *number* of packets successfully transmitted. Therefore we may assign $V = 1$ in Corollary 4, implying the following corollary:

Corollary 5. *In the case of uniform-profits, the competitive ratio of any randomized algorithm is* $\Omega(\min\{W, M\})$.

In the special case of uniform-work, we can assign $W_0 = W$ in Propositions 2 and 3, implying the following corollary:

Corollary 6. *In the case of uniform-work, the competitive ratio of any randomized algorithm is* $\Omega(\min\{V, M\})$.

2.2. Non-restricted adversaries

In Section 2.1 we assumed that the optimal algorithm has a buffer capacity of storing only one packet. This assumption significantly simplified the proofs there. In this section we relax this assumption, and show a stronger bound for the general, and more natural case, where the size of the buffer available to the optimal algorithm is identical to the size that available to the online algorithm. We use again Yao's method [27], which we used in the proof of Theorem 1. Furthermore, we use the same scenario and algorithm SubOPT, defined in Section 2.1. However, as we now allow SubOPT to store multiple packets in its buffer, SubOPT can increase its expected throughput by buffering (W_0, V)-packets whenever the number of arriving (W_0, V)-packets in a single iteration is larger than one, and processing them in iterations where no (W_0, V)-packets arrive. We now evaluate the performance in such settings.

Denote by q_j the state where there are j (W_0, V)-packets in the buffer of SubOPT at the beginning of an iteration. Note, that when $j > 0$, the count represented by q_j also includes the packet, which is to be transmitted during the iteration. Namely, SubOPT successfully transmits a packet in every iteration, unless its buffer's state is q_0.

We now turn to describe the transition matrix. Denote the probability of having exactly k (W_0, V)-packets arriving during one iteration by α_k. In each iteration we have $M \cdot W_0$ arriving packets (M packets per cycle, times W_0 cycles per iteration) which are i.i.d. where each packet is a (W_0, V)-packet with probability p. Therefore $\alpha_k = \binom{M \cdot W_0}{k} p^k (1-p)^{M \cdot W_0 - k}$ when $0 \leq k \leq MW_0$ and $\alpha_k = 0$ otherwise.

Then, the transition matrix is

$$\Pi = \left\{ \begin{array}{cccccc} \alpha_0 & \alpha_1 & \alpha_2 & \dots & \alpha_{B-1} & 1 - \sum_{j=0}^{B-1} \alpha_j \\ \alpha_0 & \alpha_1 & \alpha_2 & \dots & \alpha_{B-1} & 1 - \sum_{j=0}^{B-1} \alpha_j \\ 0 & \alpha_0 & \alpha_1 & \dots & \alpha_{B-2} & 1 - \sum_{j=0}^{B-2} \alpha_j \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & \alpha_0 & \alpha_1 & 1 - \sum_{j=0}^{1} \alpha_j \\ 0 & 0 & \dots & 0 & \alpha_0 & 1 - \alpha_0 \end{array} \right\}$$

where Π_{ij} is the probability of transition from state i to state j for each $0 \leq i, j \leq B$. Π is irreducible, because it is possible to get from any buffer state to any other buffer state by some arrival sequence. Π is also aperiodic, because its diagonal is non-zero, which represents the fact that if the buffer contains i packets at the beginning of a certain iteration, there exists a positive probability that it would contain i packets also at the beginning of the next iteration. Furthermore, as Π is finite, irreducible and aperiodic, it is also ergodic, namely, there exists a steady state. For a long enough input sequence, we can neglect the transient "warm-up" period, and assume that the expected number of iterations where SubOPT gains nothing during phase 1 is $\frac{N}{2W_0} \cdot p_0$, where p_0 is the probability that SubOPT is in state q_0. In the rest of the iterations in phase 1 SubOPT gains V per iteration. Therefore, the expected throughput of SubOPT satisfies

$$TP(\text{SubOPT}) \geq \frac{N}{2W_0} \cdot V(1 - p_0) \tag{5}$$

The expected throughput of Alg remains the same as in Eq. 2. In order to obtain the competitive ratio for the fully heterogenous case, we divide Eq. 5 by Eq. 2 and assign again $W_0 = 1$ and $p^* = \frac{1}{V(W-1)}$. Then, when $N \to \infty$ the competitive ratio is $c \geq \frac{V}{2}(W-1)(1-p_0)$.

We find p_0 by solving the balance equations defining the steady state of the system, i.e., finding the eigenvector of the transition matrix Π. Fig. 2 depicts the lower bounds as a function of M when $V = W = 10$ for various buffer sizes. Recall that the probability of a certain packet to be a (W_0, V)-packet is $p_* = \frac{1}{V(W-1)} = \frac{1}{90}$. Therefore only when M is large enough, the expected number of (W_0, V)-packets per iteration is sufficient for allowing SubOPT to

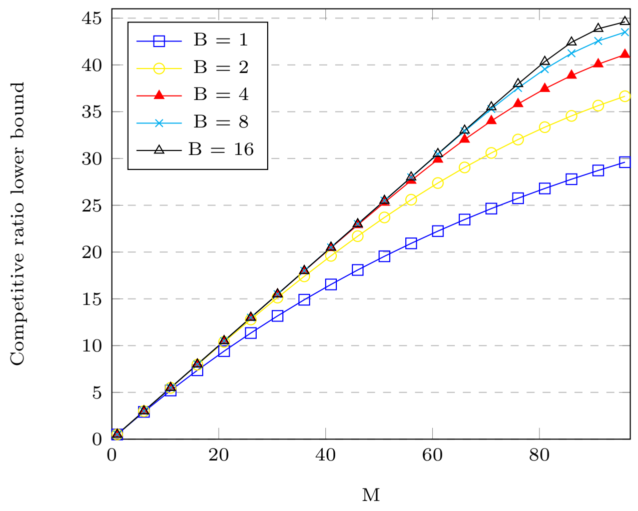


Figure 2: Lower bounds on the competitive ratio of every randomized algorithm when $W = V = 10$ where the number of unknown packets arriving in a time slot varies, for different values of B

really take advantage of its buffer for increasing its performance, resulting in a stronger lower bound on the competitive ratio.

In the next section we use the insight obtained from the analysis in the current section to identify several algorithmic concepts appropriate for the problem of buffering with limited knowledge.

3. Algorithmic Concepts

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

Random selection. For obtaining a good competitive ratio we would like to avoid a scenario where OPT successfully transmits a bulk of "good" packets, which are originally unknown, while having the online algorithm discard all these packets. This translates to assuring each arriving U-packet has some minimal probability of being accepted and parsed.

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. We refer to the act of over-prioritizing an unknown packet p in some cycle t as *admitting* p. Respectively, we refer to such a cycle t as an *admittance cycle*, and to such a packet p as an *admitted packet*.

Classify and randomly 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 Randomly Select scheme [28]. This approach is based on the following notion: Assume we have an algorithm Alg_c which is guaranteed to be c-competitive if traffic is sufficiently uniform, i.e., for cases where traffic characteristics are within some well-defined range of values. Given some arbitrary input sequence, which might be highly heterogeneous, we virtually partition the sequence of arriving packets into $N > 1$ disjoint sub-sequences, which we refer to as *classes*, such that each class is sufficiently uniform, i.e., for any specific class $1 \leq i \leq N$ the characteristics of packets corresponding to class i are within some well-defined range of values (as prescribed by Alg_c). The scheme then dictates selecting one of the classes *uniformly at random*, and applying Alg_c to this class, while ignoring all packets corresponding to other classes. One then shows that the overall competitive ratio of this randomized approach is $O(N \cdot c)$-competitive for the overall input sequence.

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. In terms of buffer management and scheduling, this translates to defining some periods, in which the algorithm processes and transmits all the packets in its buffer, even at the cost of discarding all the arrivals. If these flush periods are short enough, the algorithm gains the high throughput from flushing its buffer, yet without compromising too much throughput due to having packets discarded during the flush.

4. Competitive Algorithms

In this section we present a basic competitive online algorithm for the problem of buffering and scheduling with limited knowledge. We first provide a high-level description of our algorithm, and then turn to specify its details and analyze its performance.

For simplicity of analysis and algorithm presentation, we assume that the set of possible values of W and V – the work and profit per packet, respectively – are known to the algorithm in advance. In Sections 5 and 6 we show how to remove this assumption without harming the performance of our algorithm, and present several improved variants of this algorithm. We further note that neither of our proposed solutions require knowing the value of M – the maximum number of unknown packets arriving in a single cycle – in advance.

4.1. High-level Description of Proposed Algorithm

Our algorithm is designed according to the algorithmic concepts presented in Section 3 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 cycle chosen as an admittance cycle, the algorithm picks exactly one of the 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. We note that if no such U-packet exists, or if t is not an admittance cycle, then the algorithm may only accept known arriving packets, and would eventually schedule the top-priority packet residing in the Head-of-Line (HOL) for processing. The exact notion of priority will be detailed later.

Classify and randomly 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 (e.g., they have low profit and require much work). However, this naive description is meant only to simplify the analysis. In Section 5 we show how to remedy this naive approach in order to deal with these apparent shortcomings, 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 *Gfull* if its buffer is filled with known G-packets. Once becoming Gfull, 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 5 we show how to improve upon this naive simplified approach.

4.2. A General Classify and Randomly Select Mechanism

We now turn to explain the fundamentals of the classifying mechanism of our algorithm.

For each packet p we assign a *work-class* $C_i^{(W)}$, and denote the set of potential characteristic values within class $C_i^{(W)}$ by $X_i^{(W)}$. Let δ_W denote the maximal ratio between the work values of two packets, which belong to the same work-class. Similarly, for each packet p we assign a *profit-class* $C_i^{(P)}$, and

Algorithm 1 UpdatePhase()

```
1: if buffer is empty then
2:     $phase$ = fill
3: else if buffer is Gfull then
4:     $phase$ = flush
5: end if        ▷ if buffer is neither empty nor Gfull, $phase$ is unchanged.
```

Algorithm 2 SortBuf()

```
1: sort queued packets as follows: admitted packet first; $G^K$-packets next; rest
   of the packets last; break ties by FIFO
```

Algorithm 3 MakeRoom()

```
1: if the buffer is full then
2:     SortBuf()
3:     drop a packet from the tail
4: end if
```

denote the set of potential characteristic values within class $C_i^{(P)}$ by $X_i^{(P)}$. Let δ_V denote the maximal ratio between the profits of two packets, which belong to the same profit-class. Throughout our analysis, we will use δ_V and δ_W which are both constants.

Denote by ℓ_W and ℓ_V the number of work-classes and profit-classes, respectively. 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 $C_j^{(P)}$. Note that in terms of work, the class to which a packet p belongs is defined statically by the total work of p, and does not depend upon its remaining processing cycles, which may change over time.

Upon initialization, the algorithm selects a class by picking $i^* \in \{1, \ldots, \ell_W\}$ and $j^* \in \{1, \ldots, \ell_V\}$, each chosen uniformly at random. Then, the selected combined-class is $G = C_{(i^*, j^*)}$.

We will later define several ways to partition the packets into classes, each tailored and optimized for some specific scenarios of possible work and profit values.

4.3. The SA *Algorithm*

We now describe the details of our algorithm, Speculatively Admit (SA), depicted in Algorithm 4. The pseudo-code in Algorithm 4 uses the procedures *UpdatePhase()*, *SortBuf()*, and *MakeRoom()*, whose pseudo-code appears in Algorithms 1, 2 and 3, respectively. The procedure MakeRoom() is destined to assure a free space for a high-priority arriving packet, even at the cost of pushing-out and dropping a lower-priority packet from the tail of the buffer, if the buffer is full.

Once in the arrival step, algorithm SA updates its phase (line 1). In each cycle, the algorithm tosses a coin with some probability r, to be determined

Algorithm 4 SA: at every time slot t after transmission

```
Arrival Step:
1:  $phase$ = UpdatePhase()
2:  $admittance$ = true w.p. $r$
3:  while $phase$ == fill and exists arriving packet $p$ do
4:      if $p$ is a $G^K$-packet then
5:          if there are $B-1$ $G^K$-packets in the buffer then
6:              drop admitted packet if exists
7:          end if
8:          MakeRoom()
9:          accept $p$
10:     else if $p$ is unknown AND $admittance$ then
11:         if $A^{(U)}_{(t_p)} = 1$ then
12:             MakeRoom()
13:             mark $p$ as admitted
14:             accept $p$
15:         else
16:             w.p. $1/A^{(U)}_{(t_p)}$, swap the admitted packet with $p$.
17:         end if
18:     end if
19:     if buffer is not full then
20:         accept $p$
21:     end if
22:     $phase$ = UpdatePhase()
23:     SortBuf()
24: end while
Processing Step:
25: process HoL-packet
26: $phase$ = UpdatePhase()
27: SortBuf()
```

later, to decide whether this is an *admittance cycle*, namely, a cycle in which the algorithm may admit an unknown packet (line 2). If the phase is flush, the algorithm skips the while loop (lines 3-24), thus discarding all arriving packets.

If the phase is fill, which in particular implies that the buffer is not Gfull, the algorithm accepts every arriving G^K-packet (lines 4-9). For assuring a free slot for the arriving G^K-packet, the algorithm calls MakeRoom() (line 8) before accepting the packet (line 9). The if-clause in lines 5-7 handles the special case where there are already $B-1$ G^K-packets in the buffer; in this special case, after accepting the arriving G^K-packet, the buffer will become Gfull, and therefore it should stop admitting packets.

If the phase is fill and this is an *admittance* cycle (line 10), the algorithm admits a single U-packet arriving in this cycle, if such a packet exists. In lines 11,16, $A^{(U)}_{(t_p)}$ denotes the number of U-packets which arrive in cycle t by the arrival of packet p, including p itself. Lines 11-17 essentially perform a reservoir sampling [29], which imply that the admitted U-packet is chosen uniformly at random out of all U-packets arriving in this cycle.

Finally, if the buffer is not full, the algorithm greedily accepts every arriving packet (lines 19-20).

While in the processing step, the algorithm simply processes the top-priority packet in the buffer (line 25). Finally, the algorithm updates its phase and sorts the queued packets each time it either accepts or processes a packet (lines 22-23 and 26-27). Note that the marking of a packet as an "admitted packet" is *cycle-based*, namely, once an admitted packet is processed, it is not considered "admitted" anymore. To better understand SA, please refer to Appendix D, showing a running example of the algorithm.

4.4. Performance Analysis

We now turn to show an upper bound on the performance of our algorithm (for $W, V > 1$), captured by the following theorem (see Appendix B for the proof):

Theorem 7. SA *is* $O\left(\left[\frac{M}{r} + \delta_W \cdot \delta_V\right] \cdot \ell_W \cdot \ell_V\right)$ *-competitive.*

Theorem 7 shows an inverse linear dependency of the competitive ratio on the probability of choosing a cycle as an admittance cycle r. Thus, the best competitive ratio is attained for $r = 1$, i.e., every cycle where U-packets arrive 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, thus speeding up the processing of G^K-packets. If one indeed chooses $r = 1$, randomization should be maintained only for choosing the specific U-packet to be admitted, and the choice of the selected class. We further explore the effect of the choice of parameter r in Section 7.

In the special cases of homogeneous work values (homogeneous profit values), we assign $\delta_W = \ell_W = 1$ ($\delta_V = \ell_V = 1$, resp.) in the upper bound implied by Theorem 7, and obtain the following corollary:

Corollary 8.
(a) In the special case of homogeneous work values, SA *is* $O\left(\left(\frac{M}{r} + \delta_V\right) \cdot \ell_V\right)$*-competitive.*
(b) In the special case of homogeneous profit values, SA *is* $O\left(\left(\frac{M}{r} + \delta_W\right) \cdot \ell_W\right)$*-competitive.*

Lastly, we note that when all packets are known upon arrival, i.e. $M = 0$, SA is $(\delta_W \cdot \delta_V \cdot \ell_W \cdot \ell_V)$-competitive (see Appendix B).

4.5. Concrete Classification Mechanisms

We now show various classify and randomly select mechanisms, which are tailored and optimized for different scenarios, depending on the profit and work values.

A linear classification. When a characteristic consists of a small set of potential values, we let each class include a single value of this characteristic. As a result, the competitive ratio of the algorithm is linearly depended upon the number of distinct potential value of the respective characteristic. For instance, when the set of potential work values is small, we let each potential work value define a class. As a result, the competitive ratio of SA, implied by Theorem 7, is linearly depended upon the number of distinct work values, captured by the parameter ℓ_W. Note that in this case we have $X_i^{(W)} = \{w_i\}$, implying that δ_W, the max-to-min ratio of values within $X_i^{(W)}$, is 1.

A logarithmic classification. When the set of potential values of a characteristic is large, letting each value define a unique class results in a poor competitive ratio. Therefore, in such cases we use a logarithmic-scaled class partitioning as follows. We say that a packet p is of a certain class (either work- or profit-) i if its corresponding value is in the interval

$$X_i = \begin{cases} [1,2] & i = 1 \\ [2^{i-1}+1, 2^i] & i > 1. \end{cases} \tag{6}$$

In particular, using the above partition packets into classes, we obtain that $\delta_V = \delta_W = 2$, $\ell_V = \log_2 V$ and $\ell_W = \log_2 W$. Using Theorem 7, we obtain the following corollary:

Corollary 9. SA *is* $O\left(\frac{M}{r} \log_2 W \log_2 V\right)$*-competitive.*

We note that if we know the number of distinct values for each characteristics and the values of W and V, we can choose the appropriate classification scheme and have ℓ_W to be the minimum between $\log_2 W$, and the number of distinct work values; and have ℓ_V to be the minimum between $\log_2 V$, and the number of distinct profit values. Moreover, in any of our classification schemes, $\delta_W, \delta_V \leq 2$.

5. Improved Algorithms

Algorithm SA selects a single class uniformly at random so that the characteristics of packets on which it focuses, namely, G-packets, differ by at most a constant factor. This gives the sense of "uniformity" of traffic within the class being targeted, 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*, and prove that all our performance guarantees for SA still hold for SA*.

Class closure. Recall the partitioning of packets into classes, described in Section 4.2, namely, $\{C_{(i,j)} | i = 1, \ldots, \ell_W, j = 1, \ldots, \ell_V\}$. We let the (i, j)-*closure class* be defined as $C^*_{(i,j)} = \bigcup_{i' \leq i, j' \geq j} C_{(i',j')}$.

This definition means that the work of any packet in $C^*_{(i,j)}$ is within a ratio of at most δ_W of the work of any packet in $C_{(i,j)}$, and similarly for the profit of any packet in $C^*_{(i,j)}$. Formally, for any packets $p \in C_{(i,j)}$ and $p^* \in C^*_{(i,j)}$, $w(p^*) \leq \delta_W \cdot w(p)$ and $v(p^*) \geq \frac{v(p)}{\delta_V}$.

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. A simple substitution argument shows that thus picking $C^*_{(i,j)}$ by SA*, instead of selecting $C_{(i,j)}$ as done in SA, leaves the analysis detailed in Section 4.4 intact.

Fill during flush (pipelining). Algorithm SA was defined such that no arriving packets are ever accepted during the flush phase. This enables the partitioning of time into disjoint intervals (determined by SA's buffer being empty et the end of such an interval), and applying the comparison of performance of OPT, on the one hand, and SA, on the other hand, independently for each interval. In practice, however, allowing the acceptance of 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, the algorithm stores in the buffer packets which arrive during the flush phase, but never schedules them for processing before it successfully transmits all B packets that were stored in the buffer when it turned Gfull.

Improved scheduling. SA sorts the queued packets in G^K-first order. For simplicity of presentation, we assumed in Section 4 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). We consider specifically the following methods: (i) FIFO, (ii) W-then-V, which orders packets by a non-decreasing order of remaining work, and breaks ties by non-increasing order of profit, and (iii) non-increasing order of packet *effectiveness*, where the effectiveness of a packet is defined as its profit-to-work ratio.

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. All the *non*-admitted U-packets, however, are located at the tail of the queue, thus representing the fact that their priority is lower than that of every known packet. By applying different scheduling regimes, we obtain different flavors of SA*.

The following Theorem shows that the performance of all flavors of SA* is at least as good as the performance of SA.

Theorem 10. SA* *is* $O\left(\left[\frac{M}{r} + \delta_W \cdot \delta_V\right] \cdot \ell_W \cdot \ell_V\right)$ *-competitive.*

For the proof, see Appendix C. We study the performance of the various flavors of SA* in Section 7.

6. Practical Implementation

While presenting our basic algorithm in Section 4, we assumed for simplicity that the values of W and V – the maximal work and profit per packet, respectively – are known to the algorithm in advance. We now show how to relax these assumptions without harming the performance of our algorithms.

We refer to an algorithm implementation that does not know these values in advance as a *values-oblivious* algorithm, and to an algorithm implementation that knows the values of W and V in advance as a *values-aware* algorithm. We will show that a values-oblivious algorithm can obtain a performance which is no worse than that of a values-aware algorithm, even if the values-aware algorithm knows not only W and V, but also the concrete classes in which packets will arrive.

Our implementation of a values-oblivious algorithm is based on an application of reservoir sampling [29] on classes revealed during packet arrivals, as we will detail shortly. A new class is revealed either due to the arrival of a K-packet p, or due to a U-packet q being parsed, corresponding to a class previously unknown to the algorithm. We call such an event an *uncovering of a new class*.

The values-oblivious algorithm implementation performs the following alongside all decisions made by the values-aware algorithm: Before the arrival sequence begins we initiate a counter N of known classes to be $N = 0$. Upon the uncovering of a new class at t the algorithm increments N by one (to reflect the updated number of known classes), and replaces the previously selected class with the new class with probability $1/N$.

As the above procedure essentially performs a reservoir sampling on the collection of classes known to the algorithm, it essentially implements the selection of a class uniformly at random among all *a posteriori* known classes [29].

It therefore follows that the distribution of the packets corresponding to the eventual selected class (after the sequence ends) handled by the values-oblivious algorithm is identical to the distribution of the packets handled by the values-aware algorithm. Therefore the expected performance of the values-oblivious algorithm is lower bounded by the expected performance of the values-aware algorithm. We note that the implementation of the values-oblivious algorithm can be applied to any of the variants described in our previous sections.

7. 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. Our choice of distributions for the parameters of the traffic characteristic enables us to evaluate our algorithms performance in a wide range of settings.

These choices, as we show in the sequel, are also motivated by the properties of real-world traffic.

7.1. 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 burst during the HIGH state generates an average of 10 packets per cycle, while the LOW state generates an average of only 0.5 packet per cycle. The average duration of LOW-state periods is a factor W longer than the average duration of HIGH-state periods. This is targeted at allowing some traffic arriving during the HIGH-state to be drained during the LOW-state.

In our simulations, 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 $\alpha \in [0, 1]$. We thus obtain that the expected number of U-packets per cycle during the HIGH state is 10α.

In real-life scenarios, the maximum work, W, required by a packet, is highly implementation-depended. It depends on the specific hardware, processing elements, and software modules. However, several works which investigated the required work on typical tasks [30, 31, 32] indicate that W is two orders of magnitude larger than the work required for doing an IPv4-trie search or classification of a packet. We refer to IPv4-trie search or classification of a packet as the baseline unit of work, captured by our notion of "parsing". We therefore set the maximum work required by a packet to $W = 256$ throughout this section. As the potential set of characteristics is large, we use a logarithmic classification scheme (recall Section 4.5).

Determining the maximum profit, V, associated with a packet, is a challenging task. This value 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 $V = 16$ throughout this section.

The values $W = 256$ and $V = 16$ imply a total of $8 \cdot 4 = 32$ potential classes for the algorithm to select from, respectively. The value of each characteristic for each packet is drawn from an approximation of a Pareto-distribution as follows. First, we randomly generate numbers, following a Pareto-distribution. Next, numbers are rounded, to get integer values. Finally, for disallowing values above the maximum (256 for work values and 16 for profit values), all the cases where the randomly generated values were above the maximum were truncated, namely, treated as if the generated value was exactly the maximal value. The averages and standard deviations of the values obtained after this generation

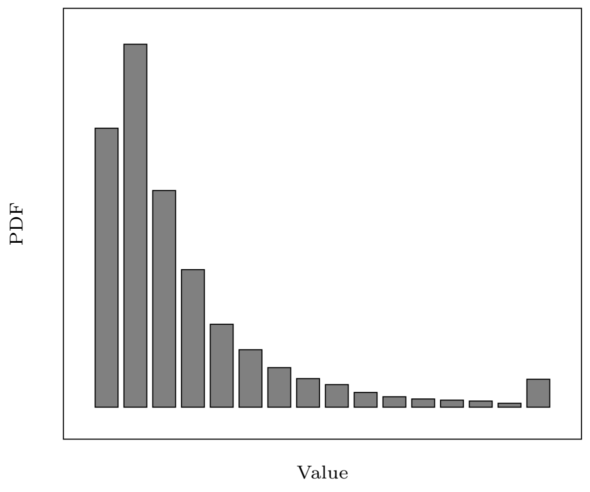


Figure 3: Probability distribution function of the characteristics values

process are 17.97 and 22.22 for packet work, and 3.66 and 3.20 for packet profit. The schematic probability distribution function of the characteristics values is depicted in Fig. 3. Note the spike at its maximum, due to the truncation described above. Unless stated otherwise, 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 (corresponding to the packet's work) and value (corresponding to the packet's profit). 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 employ the classic 2-approximation greedy algorithm for solving the knapsack problem [33], and use its performance as an approximate upper bound on the performance of OPT. For considering the additional profit which OPT may 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 4.

3. SA^* *FIFO*: Algorithm SA^* where priority ties are broken by FIFO order.

4. SA^* *W-Then-V*: Algorithm SA^* where priority ties are broken in non-decreasing order of remaining work, and further ties are broken in non-increasing order of profit. This variant is denoted by $\text{SA}^* W - V$ in Figures 4-7.

5. SA^* *EFFECT*: Algorithm SA^* where priority ties are broken in non-increasing order of their profit-to-work ratio.

We recall that all the flavors of SA^* listed above maintain a G^K-first order, and differ only in the internal ordering *within* each set (namely, within the set of G^K-packets, as well as within the set of *non*-G^K-packets).

All flavors of SA^* described above employ the class-closure and the fill-during-flush modifications defined in Section 5. For each choice of parameters we show the average of running 100 independently-generated traces of 10K packets each. In all our simulations the standard deviation was below 0.035.

7.2. Simulation Results

Figures 4-7 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 scheduling policy is by non-increasing effectiveness, followed by employing the W-then-V approach. 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 Gfull. 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.

7.2.1. The Effect of Selected Class

Our first set of results sheds light on the effect of the class selected by an algorithm on its performance. Fig. 4 shows the results where the selected profit-class j^* 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 Gfull. 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 Gfull 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; recall that SA over-prioritizes only packets which belong to a single randomly selected class, i.e., SA does not employ the class closure optimization (described in Section 5). In such a case, SA is coerced into processing also packets which do not belong to the selected class – namely, 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 Fig. 5, 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 and processes all incoming packets in FIFO order).

In subsequent results described hereafter, we fix both the work-class and the profit-class to be 3, which represents a mid-range class for both the profit and the work.

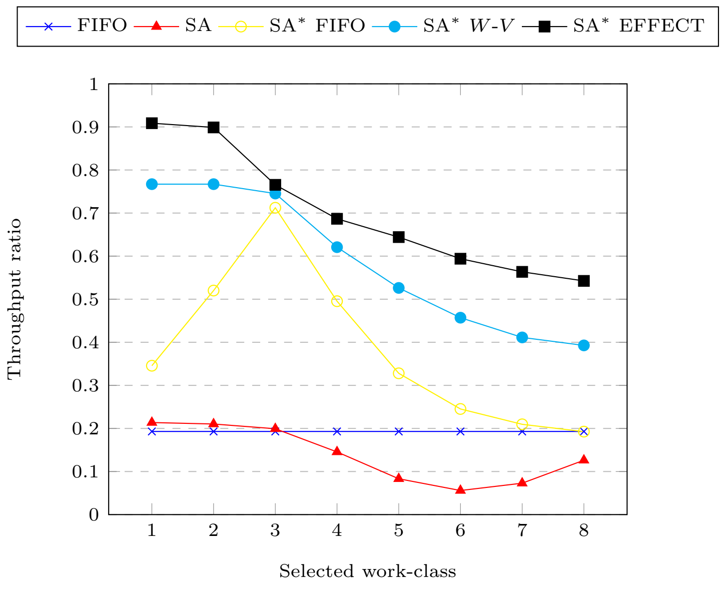


Figure 4: Effect of chosen work-class i^*

7.2.2. The Effect of Missing Information

Fig. 6 illustrates the performance ratio of our algorithms as a function of the expected number of U-packets arriving during the HIGH state, where we vary the value of α from 0 to 1. This provides further insight as to the performance of each algorithm as a function of the intensity of unknown packets. We recall that for our choice of parameters, the values of α translate to having the expected number of unknown packets per cycle during the HIGH state vary from 0 to 10. As one could expect, the performance ratio of SA and of all versions of SA* degrades as the amount of uncertainty increases.

Finally, we study the intensity of exploring unknown packets, as depicted by the choice of parameter r which determines whether a cycle is an admittance cycle or not. The results depicted in Fig. 7 consider the case of high uncertainty, where $\alpha = 1$, that is, all arriving packets are unknown.

Observe first the special case where $r = 0$, which represents an extreme case, in which, although all arriving packets are unknown, our algorithms do not explore any new packets, and actually degenerate to a simple FIFO, and therefore exhibit identical performance. Increasing the admittance probability r, however, yields a steady increase in performance, albeit with diminishing returns. Similar results were obtained also when some of the packets are known, but with smaller marginal benefits. These results coincide with our analytic results, which further validate our algorithmic approach.

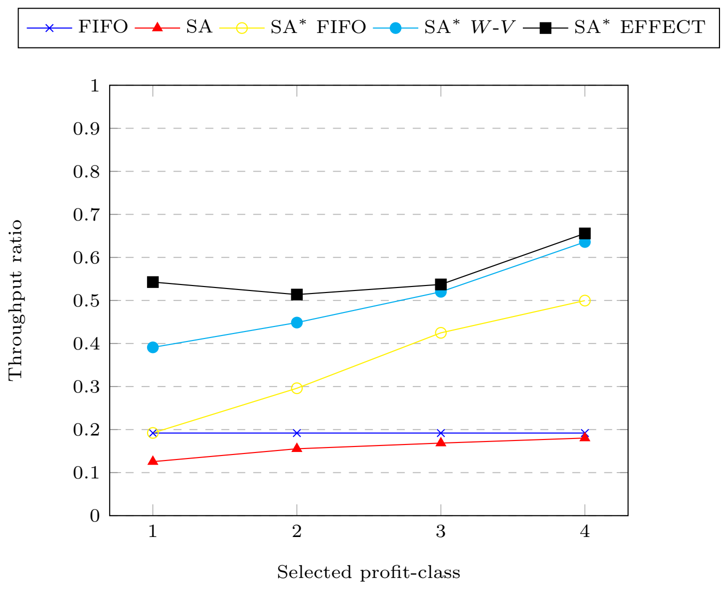


Figure 5: Effect of chosen profit-class j^*

8. Conclusions and Future Work

We consider the problem of managing buffers where traffic has unknown characteristics, namely required processing and profits. We show lower bounds on the competitive ratio of any online algorithm for the problem. We define several algorithmic concepts targeted at such settings, and develop several algorithms that follow our suggested prescription. Our theoretical analysis shows that the competitive ratio of our algorithms is not far from the best competitive ratio any online algorithm can achieve. We validate the performance of our algorithms via simulation which further serves to elucidate our design criteria. Our work can be viewed as a first step in developing fine-grained algorithms handling scenarios of limited knowledge in networking environments for highly heterogeneous traffic.

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.

References

References

[1] I. Cohen, G. Scalosub, Queueing in the mist: Buffering and scheduling with limited knowledge, in: IWQoS, IEEE, 2017, pp. 1–6.

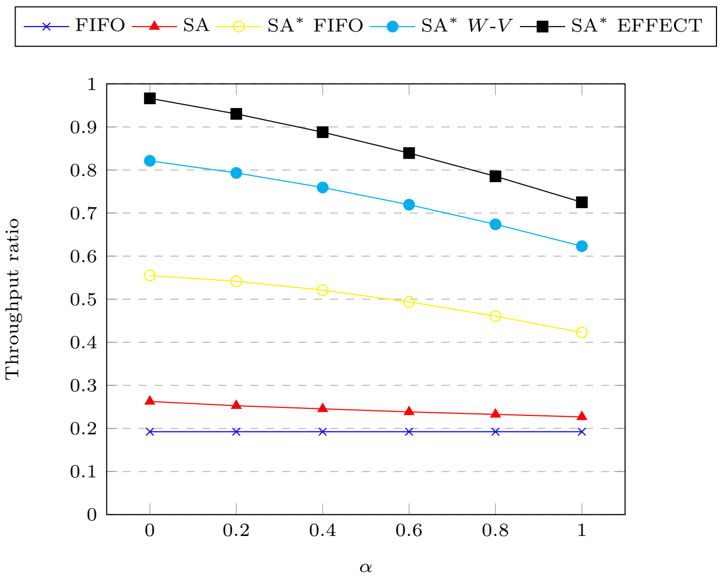


Figure 6: Effect of expected number of U-packets during the HIGH state

[2] K. Karras, T. Wild, A. Herkersdorf, A folded pipeline network processor architecture for 100 gbit/s networks, in: ANCS, 2010, p. 2.

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

[4] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, A. Vahdat, Portland: a scalable fault-tolerant layer 2 data center network fabric, in: SIGCOMM, Vol. 39, 2009, pp. 39–50.

[5] M. Yu, J. Rexford, M. J. Freedman, J. Wang, Scalable flow-based networking with difane, in: SIGCOMM, Vol. 41, 2011, pp. 351–362.

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, S. Shenker, Rethinking enterprise network control, IEEE/ACM Transactions on Networking 17 (4) (2009) 1270–1283.

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

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

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

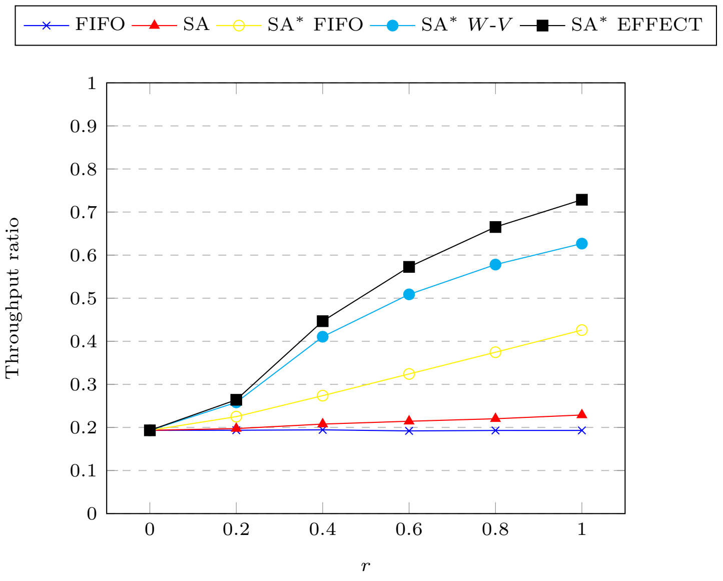


Figure 7: Effect of admittance probability of U-packets r

[10] A. Borodin, R. El-Yaniv, Online computation and competitive analysis, Cambridge University Press, 2005.

[11] W. A. Aiello, Y. Mansour, S. Rajagopolan, A. Rosén, Competitive queue policies for differentiated services, in: INFOCOM, Vol. 2, 2000, pp. 431–440.

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

[13] Y. Mansour, B. Patt-Shamir, O. Lapid, Optimal smoothing schedules for real-time streams, in: PODC, 2000, pp. 21–29.

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

[15] Y. Azar, Y. Richter, An improved algorithm for cioq switches, in: ESA, 2004, pp. 65–76.

[16] A. Kesselman, K. Kogan, M. Segal, Packet mode and qos algorithms for buffered crossbar switches with fifo queuing, Distributed Computing 23 (3) (2010) 163–175.

[17] Y. Kanizo, D. Hay, I. Keslassy, The crosspoint-queued switch, in: INFOCOM, 2009, pp. 729–737.

[18] A. Kesselman, B. Patt-Shamir, G. Scalosub, Competitive buffer management with packet dependencies, Theoretical Computer Science 489–490 (2013) 75–87.

[19] Y. Mansour, B. Patt-Shamir, D. Rawitz, Overflow management with multipart packets, Computer Networks 56 (15) (2012) 3456–3467.

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

[21] I. Keslassy, K. Kogan, G. Scalosub, M. Segal, Providing performance guarantees in multipass network processors, IEEE/ACM Transactions on Networking 20 (6) (2012) 1895–1909.

[22] K. Kogan, A. López-Ortiz, S. Nikolenko, G. Scalosub, M. Segal, Balancing work and size with bounded buffers, in: COMSNETS, 2014.

[23] Y. Azar, O. Gilon, Buffer management for packets with processing times, in: ESA, 2015, pp. 47–58.

[24] Y. Azar, I. R. Cohen, I. Gamzu, The loss of serving in the dark, in: STOC, 2013, pp. 951–960.

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

[26] K. Pruhs, Competitive online scheduling for server systems, ACM SIGMETRICS Perf. Eval. Review 34 (4) (2007) 52–58.

[27] A. C.-C. Yao, Probabilistic computations: Toward a unified measure of complexity, in: FOCS, 1977, pp. 222–227.

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

[29] J. S. Vitter, Random sampling with a reservoir, ACM Transactions on Mathematical Software 11 (1) (1985) 37–57.

[30] R. Ramaswamy, N. Weng, T. Wolf, Analysis of network processing workloads, Journal of Systems Architecture 55 (10) (2009) 421–433.

[31] M. E. Salehi, S. M. Fakhraie, Quantitative analysis of packet-processing applications regarding architectural guidelines for network-processing-engine development, Journal of Systems Architecture 55 (7) (2009) 373–386.

[32] M. E. Salehi, S. M. Fakhraie, A. Yazdanbakhsh, Instruction set architectural guidelines for embedded packet-processing engines, Journal of Systems Architecture 58 (3) (2012) 112–125.

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

Appendix A. Preliminaries

We now define some of the notation that will be used throughout the appendix.

For every cycle t and packet type α, we denote by $A^{\alpha}(t)$ the number of α-packets that arrive in cycle t. For instance, $A^{(K)}(t)$ ($A^{(U)}(t)$) denotes the number of K-packets (U-packets) which arrive in cycle t. This notation can be combined with the work and profit values of packets. For instance, $A^{(U)}_{(w,v)}(t)$ denotes the number of U-packets with work w and profit v, which arrive in cycle t.

Our proofs involve a careful analysis of the expected profit of our algorithms from packets which arrive when it is either in the fill or the flush phase. Therefore, we now turn to define the exact notion of cycles belonging to either phase. We say that an algorithm is in the flush phase in a specific cycle t if it is in the flush state at the end of the arrival step of cycle t. If it's not in the flush phase in cycle t, then we say it is in the fill phase in cycle t. Denote by $P^{(\text{fill})}$ and $P^{(\text{flush})}$ the sets of cycles in which our algorithm is in the fill and flush phases, respectively.

For every packet type α, we denote by $S_{\alpha}(t)$ the expected profit of the algorithm from α-packets which *arrive* in cycle t, and by $S_{\alpha} = \sum_t S_{\alpha}(t)$ the overall expected profit of Alg from α-packets. We denote by O_{α} the expected profit of some optimal solution, OPT, from α-packets. Again, these notations can be combined with previous notations. For instance, $O_{G^U}(t)$ denotes the overall expected profit of OPT from G^U-packets. Furthermore, $O^{(\text{fill})}_{G^U}$ denotes the expected profit of OPT from G^U-packets which arrive during $P^{(\text{fill})}$.

Appendix B. Proof Of Theorem 7

Our proof will follow from a series of propositions. Initially, we aim to prove that SA successfully transmits every G^K-packet which arrives during the fill phase, by showing that it never drops such a packet once it is accepted to the buffer.

Proposition 11. SA *successfully transmits every G^K-packet which arrives during the fill phase.*

Proof. We first note, that any G^K-packet arriving during the fill phase (depicted by the while loop in lines 3-24) is accepted (line 9).

Next, we show that SA never drops a G^K-packet which resides in its buffer. We consider all cases where SA drops a packet from its buffer, and prove that it cannot be a G^K-packet.

In line 6, SA drops an admitted packet, namely, a picked U-packet, and not a G^K-packet.

In line 8, SA performs the MakeRoom() procedure, which may result in dropping the last packet in the buffer. However, as this line dwells within the while loop of lines 3-24, we know that the phase is fill, and therefore there are at most $B-1$ G^K-packets in the buffer. Furthermore, if there are exactly $B-1$ G^K-packets in the buffer, the if-clause in lines 5-7 assures that there is no admitted packet in the buffer. Hence, if the buffer is full, it contains at least one low-priority packet – namely, a packet which is not admitted and not a G^K-packet. After sorting the buffer, this low-priority, non-G^K packet, will be located in the tail of the queue and dropped.

SA may perform the MakeRoom() procedure also in line 12, if $A^{(U)}_{(t_p)} = 1$. In this case, the arriving packet p is the first U-packet arriving in this cycle – and it is not admitted yet. As a result, there is no admitted packet in the buffer. Furthermore, as this line is executed during the fill phase (the while loop of lines 3-24), there are at most $B-1$ G^K-packets in the buffer. Hence, if the buffer is full, it contains at least one low-priority, non-G^K-packet, which is the packet dropped. □

The following lemma shows that the overall number of G-packets transmitted by SA is at least a significant fraction of the number of G-packets accepted by an optimal policy during a fill phase.

Lemma 12. $S_G \geq \frac{r}{M} O^{(\text{fill})}_G$.

Proof. Let t denote a cycle in the fill phase, in which U-packets arrive. Then, with probability r SA admits one U-packet, denoted p. As the algorithm implements reservoir sampling [29], p is picked uniformly at random out of at most M unknown arrivals, and therefore the probability that $p \in G^U$ is at least $A_{G^U}(t)/M$. As p is parsed in the cycle of arrival, in the subsequent cycle it is known. By Proposition 11, if p is a G^K-packet, then SA will eventually transmit p. Recalling that $X^{(W)}_{i^*}$ and $X^{(P)}_{j^*}$ denote the ranges of the work and profit values within the selected work and profit class $C_{(i^*,j^*)}$ (see Sec. 4.5), we conclude that

$$S_{G^U}(t) \geq \frac{r}{M} \sum_{w \in X^{(W)}_{i^*}, v \in X^{(P)}_{j^*}} [v \cdot A^{(U)}_{(w,v)}(t)]. \tag{B.1}$$

Summing Eq. B.1 over all the cycles within the fill phase,

$$S_{G^U} \geq \frac{r}{M} \sum_{t \in P^{(\text{fill})}} \sum_{w \in X^{(W)}_{i^*}, v \in X^{(P)}_{j^*}} [v \cdot A^{(U)}_{(w,v)}(t)] \geq \frac{r}{M} O^{(\text{fill})}_{G^U}. \tag{B.2}$$

In addition, by Proposition 11, $S_{G^K} \geq O^{(\text{fill})}_{G^K}$. Therefore

$$S_G = S_{G^K} + S_{G^U} \geq \frac{r}{M}(O^{(\text{fill})}_{G^K} + O^{(\text{fill})}_{G^U}) = \frac{r}{M} O^{(\text{fill})}_G. \tag{B.3}$$

□

We are now in a position to prove Theorem 7.

Proof of Theorem 7. Every class $C_{(i,j)}$ is the selected class with probability $\frac{1}{\ell_W \cdot \ell_V}$. Using Lemma 12 we therefore have for all $i \in \{1, 2, \ldots, \ell_W\}$ and $j \in \{1, 2, \ldots, \ell_V\}$, $S_{(i,j)} \geq \frac{r}{M \cdot \ell_W \cdot \ell_V} O^{(\text{fill})}_{(i,j)}$.

Summing over all the classes, we obtain that the expected performance of our algorithm satisfies

$$\sum_{i=1}^{\ell_W} \sum_{j=1}^{\ell_V} S_{(i,j)} \geq \frac{r}{M \cdot \ell_W \cdot \ell_V} \sum_{i=1}^{\ell_W} \sum_{j=1}^{\ell_V} O^{(\text{fill})}_{(i,j)}. \tag{B.4}$$

If SA is never Gfull during an arrival sequence, then $O_{(i,j)} = O^{(\text{fill})}_{(i,j)}$ and therefore, by Eq. B.4 the ratio between the performance of OPT and the expected throughput of SA is at most $\frac{M}{r} \cdot \ell_W \cdot \ell_V$, as required.

Assume next that SA becomes Gfull during an input sequence. In such a case we compare the overall throughput due to packets *transmitted* by SA until the first cycle in which its buffer is empty again, and the profit obtained by OPT due to packets *accepted* by OPT during the same interval. We note that our analysis would also apply to subsequent such intervals, namely, until the subsequent cycle in which SA is empty again.

We note that in case SA becomes Gfull, SA holds in its buffer exactly B G-packets, and all these packets are transmitted by the time SA is empty again. By the definition of δ_W in Section 4.2, the maximal work which SA dedicates to any of these packets is at most δ_W times higher than the *minimal* work which OPT dedicates to any G-packet. As a result, during the flush phase, in which SA handles B G-packets, OPT can handle at most $\delta_W B + B$ G-packets. Furthermore, by the definition of δ_V in Section 4.2, the maximal profit of OPT from any G-packet is at most δ_V higher than the *minimal* profit of SA from any G-packet. Combining the above reasoning implies that

$$\frac{O^{(\text{flush})}_G}{S_G} \leq \frac{\delta_W B + B}{B} \cdot \delta_V = (\delta_W + 1)\delta_V. \tag{B.5}$$

As every class $C_{(i,j)}$ is the selected class w.p. $\frac{1}{\ell_W \cdot \ell_V}$, we have

$$\forall i \in \{1 \ldots \ell_W\}, j \in \{1 \ldots \ell_V\}, S_{(i,j)} \geq \frac{1}{(\delta_W + 1)\delta_V \cdot \ell_W \cdot \ell_V} O^{(\text{flush})}_{(i,j)}.$$

Summing over all the classes we obtain

$$\sum_{i=1}^{\ell_W} \sum_{j=1}^{\ell_V} S_{(i,j)} \geq \frac{1}{(\delta_W + 1)\delta_V \cdot \ell_W \cdot \ell_V} \sum_{i=1}^{\ell_W} \sum_{j=1}^{\ell_V} O^{(\text{flush})}_{(i,j)}. \tag{B.6}$$

Combining Equations B.4 and B.6 implies that the competitive ratio of SA is at most

$$\frac{\sum_{i=1}^{\ell_W} \sum_{j=1}^{\ell_V} \left[O^{(\text{fill})}_{(i,j)} + O^{(\text{flush})}_{(i,j)} \right]}{\sum_{i=1}^{\ell_W} \sum_{j=1}^{\ell_V} S_{(i,j)}} \leq \left[\frac{M}{r} + (\delta_W + 1)\delta_V \right] \cdot \ell_W \cdot \ell_V, \tag{B.7}$$

which completes the proof. □

In the special case where all packets are known upon arrival, we obtain the following upper bound on the competitive ratio of SA:

Corollary 13. *When $M = 0$,* SA *is $O(\delta_W \cdot \delta_V \cdot \ell_W \cdot \ell_V)$-competitive.*

Proof. We follow the proof of Theorem 7, and carefully check the required changes.

When all packets are known, Proposition 11 remains essentially intact. Furthermore, we have $S_G = S_{G^K} \geq O^{(\text{fill})}_{G^K} = O^{(\text{fill})}_G$, which replaces Lemma 12. Accordingly, Eq. B.4 is modified to

$$\sum_{i=1}^{\ell_W}\sum_{j=1}^{\ell_V} S_{(i,j)} \geq \frac{1}{\ell_W \cdot \ell_V}\sum_{i=1}^{\ell_W}\sum_{j=1}^{\ell_V} O^{(\text{fill})}_{(i,j)}. \qquad \text{(B.8)}$$

Equation B.6 remains intact, as in deriving it we use the classify and randomly select scheme, independently of M. Combining Equations B.8 and B.6 implies that when all packets are known, the competitive ratio of SA is at most

$$\frac{\sum_{i=1}^{\ell_W}\sum_{j=1}^{\ell_V}\left[O^{(\text{fill})}_{(i,j)} + O^{(\text{flush})}_{(i,j)}\right]}{\sum_{i=1}^{\ell_W}\sum_{j=1}^{\ell_V} S_{(i,j)}} \leq \left[1 + (\delta_W + 1)\delta_V\right] \cdot \ell_W \cdot \ell_V, \qquad \text{(B.9)}$$

which completes the proof. □

Appendix C. Proof Of Theorem 10

Proof. We first consider the effect of uniformly at random selecting a class closure, instead of selecting a specific class. First, note that the proof of Lemma 12 also directly applies to SA*, implying that $S^*_{G^*} \geq \frac{r}{M} O^{(\text{fill})}_G$. Furthermore, the arguments used in the proof of Theorem 7 also apply to SA*, and in particular SA* satisfies Equation B.7, where we substitute in the denominator $S_{(i,j)}$ by $S^*_{(i,j)}$.

Consider next the affect of performing fill during flush. In SA* we accept packets also during the flush phase, but we never process any of these packets before all packets contributing to the algorithm being Gfull are transmitted, i.e., they are never processed before the flush phase is complete. We enumerate the fill phases and the subsequent flush phases as follows: $P_{\text{fill}_1}, P_{\text{flush}_1}, P_{\text{fill}_2}, P_{\text{flush}_2}, \ldots, P_{\text{fill}_n}, P_{\text{flush}_n}$, where $n \geq 1$. It should be noted that each such phase corresponds to a series of disjoint time intervals defined by the first cycle of the sequence of phases. We further denote the P_{flush_0} phase as an empty set of cycles, and in case that the sequence ends by a fill phase, we also let P_{flush_n} denote an empty set of cycles. Similarly, we further define $P^*_{\text{fill}_i}, P^*_{\text{flush}_i}$, for the appropriate values of i, to denote the fill and flush phases corresponding to SA*.

Denote the profit accrued by SA and OPT from packets which arrive during the i^{th} fill phase by $S^{(P_{\text{fill}_i})}$ and $O^{(P_{\text{fill}_i})}$ respectively. Similarly, denote the profit of SA and OPT obtained from packets which arrive during the i^{th} flush phase by $S^{(P_{\text{flush}_i})}$ and $O^{(P_{\text{flush}_i})}$, respectively. Similarly, we let $S^{*(P^*_{\text{fill}_i})}$ and $S^{*(P^*_{\text{flush}_i})}$ indicate the profit of SA* obtained from packets which arrive during its i^{th} fill and flush phase, respectively.

Using this notation, we recall that, by the analysis of SA presented in Theorem 7

$$O^{(P_{\text{fill}_i})} + O^{(P_{\text{flush}_i})} \leq \left[\frac{M}{r} + (\delta_W + 1)\delta_V\right] \ell_W \cdot \ell_V \cdot S^{(P_{\text{fill}_i})} \tag{C.1}$$

for every $i = 1, \ldots, n$.

This induces an implicit mapping ϕ of the units of profit obtained from G-packets accepted by OPT during $P_{\text{fill}_i} \cup P_{\text{flush}_i}$ to the units of profit obtained from G-packets accepted by SA during P_{fill_i} (either known, or unknown that were parsed), such that every unit of profit obtained by SA has at most $\left[\frac{M}{r} + (\delta_W + 1)\delta_V\right] \ell_W \cdot \ell_V$ units of profit mapped to it.

A key observation is noting that the image of mapping ϕ is essentially the profit attained from the set of G-packets contributing to the algorithm being Gfull at the end of the corresponding fill phase.

As SA* may accept packets during flush, in the beginning of the subsequent fill phase the buffer of SA* may not be empty. In particular, there could be G-packets accepted during the recent flush phase that are stored in the buffer. However, none of these packets have any OPT packets mapped to them. It follows that these packets can contribute to SA* becoming Gfull in the new fill phase, and any profit implicitly mapped to the profit of these packets by ϕ would correspond to packets arriving during the new fill phase, or its subsequent flush phase. The implicit mapping is depicted in Fig. C.8, along with the difference between the mapping arising from the behavior of SA (visualized above the time axis), and the mapping arising from the behavior of SA*(visualized below the time axis). Note that the fill and flush phases of both algorithms need not be synchronized, since SA* can potentially become Gfull "faster" than SA.

It follows that Eq. C.1 now translates to

$$\begin{aligned} O^{(P^*_{\text{fill}_i})} + O^{(P^*_{\text{flush}_i})} &\leq \\ &\left[\tfrac{M}{r} + (\delta_W + 1)\delta_V\right] \ell_W \cdot \ell_V \cdot \left[S^{*(P^*_{\text{flush}_{i-1}})} + S^{*(P^*_{\text{fill}_i})}\right] \end{aligned} \tag{C.2}$$

for every $i = 1, \ldots, n$. Summing over all $i = 1, \ldots, n$, we obtain that the competitive ratio guarantee for SA* is the same as that for SA.

Lastly, the analysis of SA does not assume any specific scheduling rule to be applied, as long as the $G^{(K)}$-first order rule is maintained. Thus, our competitive ratio guarantee is independent of the specific ordering within the set of G^K-packets, as well as within the set of non-G^K-packets. □

Appendix D. Running Example of SA

Figure D.9 exemplifies a running of SA equipped with a 3-slots buffer. Each packet is represented by a square. If it is a known (w, v)-packet, then (w, v)

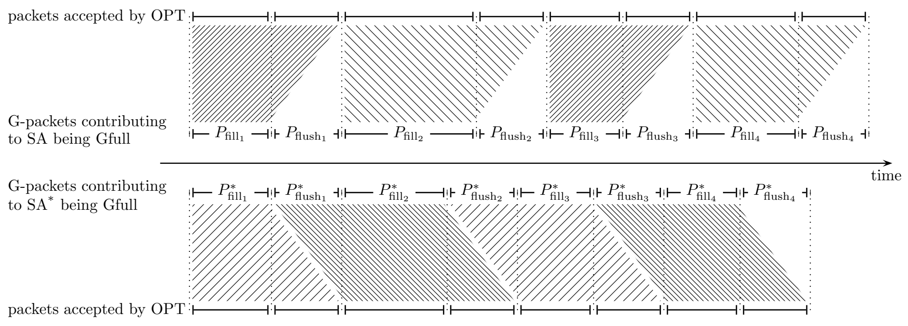


Figure C.8: Visualization of the mappings induced by the analysis of SA and SA*, for the first 4 fill and flush phases. The fill and flush phases of SA are denoted P_{fill_i} and P_{flush_i}, respectively, whereas the fill and flush phases of SA* are denoted $P^*_{\text{fill}_i}$ and $P^*_{\text{flush}_i}$, respectively. The top part shows the mapping of profit corresponding to packets accepted by OPT along time, to the profit corresponding to G-packets accepted by SA during the fill phase (since SA does not accept any packets during the flush phase). The bottom part shows the induced mapping of profit obtained by packets accepted by OPT along time to the profit of G-packets accepted by SA* during both the preceding flush phase, and the current fill phase.

(namely, its work, profit values, resp.) appears within the square representing the packet. If the packet is unknown, however, the (unknown) work and profit values do not appear, and the packet's color is dark gray.

Known packets which belong to the selected class (G^K-packets) are marked in light gray. The figure assumes that the (randomly-chosen) selected class is the class of packets with work- and profit- values within the range [3, 4]. Recall, that this range refers to the characteristics of a packet *upon arrival*. For instance, a $(3, 3)$-packet always belongs to the selected class, although after being processed its residual work decreases, and it becomes a $(2, 3)$-packet, and later a $(1, 3)$-packet, and so on.

Each cycle begins with the transmission step, in which a fully processed packet, if such exists, leaves the queue. In our example there is no packet transmitted since we focus our attention on handling arrivals and determining priorities which are the core components of our algorithm. This step is followed by the arrival step, where arriving packets are handled by the algorithm. Finally, the cycle ends with a processing step, where the head-of-line (HoL) packet is processed. This packet is emphasized by an extra internal square. The state of the queue at the end of each cycle is depicted by a light-gray background. At each cycle the algorithm tosses a coin, and assigns the cycle as an *admittance cycle* w.p. r. In this example, we assume that cycles $1, 3, 5, 6$ are admittance cycles. We now turn to explain the scenario depicted in Figure D.9 cycle by cycle.

$t = 0$. Begin with an empty buffer.

$t = 1$. A known $(4, 4)$-packet arrives. As both its work- and profit- values belong to the ranges [3,4], it is a G^K-packet, and therefore it is retained by the algorithm (recall that G^K-packets are never dropped during the fill phase, as shown in Proposition 11).

Next, a U-packet arrives. As this is an admittance cycle, this U-packet is *admitted*, that is, accepted into the buffer, and assigned to the HoL. Since this is the last packet to arrive in this cycle, and being the HoL-packet, this packet is processed in the processing step. We refer to this packet as being *parsed* in this cycle, as this is the first processing cycle of this packet.

After parsing, the characteristics of the HoL packet become known: it is now a known $(1, 8)$-packet. Namely, when it arrived, it was a $(2, 8)$-packet which has received one cycle of processing. By these values, this packet does not belong to the selected class. Therefore, it is pushed down to the buffer's tail. Instead, the G^K-packet, with values $(4, 4)$ is assigned to be the HoL packet. It should be noted that although the parsed $(1, 8)$-packet is superior to any G^K-packet currently in the buffer (since it carries a profit value of 8 while requiring just one more cycle of work), SA still prefers G^K-packets over this packet. We note that the improved SA* algorithm would re-assign such a packet to be a G^K-packet by considering the selective class closure.

$t = 2$. No packets arrive. The HoL-packet, $(4, 4)$, is processed, and becomes a $(3, 4)$-packet.

$t = 3$. This is an admittance cycle. Therefore, the first arriving U-packet is admitted. In particular, this cycle well exemplifies the buffer's ordering: at top-priority is the admitted packet; at a second priority is the G^K-packet, $(3, 4)$; the remaining packet in the buffer, $(1, 8)$, is of a lowest priority.

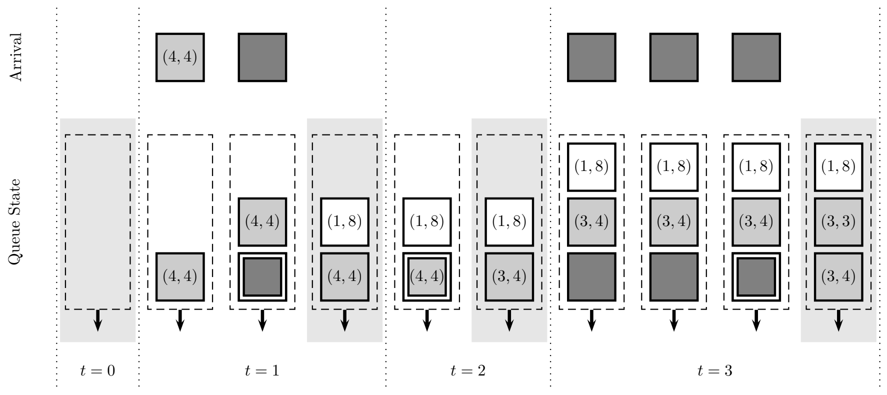
When a second U-packet arrives, SA tosses a coin, and replaces the previously-admitted packet with the new arriving U-packet w.p. $1/2$. When a third U-packet arrives, SA tosses a coin again, and replaces the previously-admitted packet with the new arriving U-packet w.p. $1/3$.

In the processing step, SA parses the admitted packet, unraveling it as a $(3, 3)$-packet. Namely, upon arrival its characteristics were $(4, 3)$, ascribing it to the selected class. As there already exists another G^K-packet in the buffer (the $(3, 4)$-packet) SA breaks the tie between the two G^K-packets in its buffer by FIFO order. We note that the improved SA* algorithm would transition to the flush phase at this point, since it would have been full of G^K-packets.

$t = 4$. First, we have an arriving known $(2, 5)$-packet. By its characteristics, it is not a G^K-packet. Therefore, it is assigned the lowest priority. In particular, as the buffer is full, this packet is discarded. Next, a U-packet arrives. However, as this is a non-admittance cycle, the U-packet is discarded as well. Finally, during the processing step, the HoL packet is processed, decreasing its remaining work to 2.

$t = 5$. We have a single arriving U-packet. As it is an admitted cycle, this U-packet is admitted, hence, accepted and parsed. In order to make room for this admitted packet, the $(1, 8)$-packet in the tail is pushed-out and dropped. After parsing, the U-packet is uncovered as a $(1, 2)$-packet. Namely, upon arrival it was a $(2, 2)$-packet. By these characteristics, this packet does not belong to the selected class, and therefore has the lowest priority, and downgraded to the tail.

$t = 6$. This is an admittance cycle. Therefore the first arriving U-packet is admitted, pushing-out from the buffer the $(1, 2)$-packet, which was in the tail. When a second U-packet arrives, it replaces the previously-admitted packet w.p. 1/2. Then, a $(2, 7)$-packet arrives. By its characteristics, it is neither an admitted packet (as it is a K-packet), nor does it belong to the selected class. As a result, the $(2, 7)$-packet is assigned the lowest priority, and is therefore discarded. The last arrival in this cycle is a known $(4, 4)$-packet. By its characteristics, it is a G^K-packet. Since the buffer already contains $B - 1 = 2$ G^K-packets, the U-packet at the HoL is dropped, and the newly arriving CsK-packet is accepted to the queue (see lines 4-9 in Algorithm 4). The queue therefore becomes Gfull, i.e., the buffer is full with G^K-packets. SA then switches to the *flush* state, and it will merely process all the packets in its buffer in a run-to-completion manner and transmit all the fully-processed packets, until the buffer is empty again.



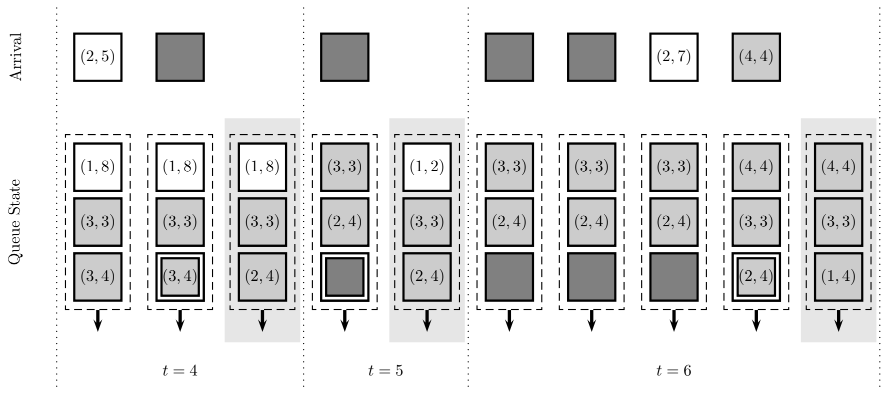


Figure D.9: Running Example of SA, equipped with a 3-slots buffer. Each known packet is labeled $(w(p), v(p))$, where w is the remaining work and v is the profit. G^K-packets are marked by light gray. U-packets are colored by dark gray.
Each cycle begins with a transmission step, in which a fully-processed packet, if such exists, is transmitted. Next comes the arrival step, where arriving packets are handled by the algorithm one by one. For each arriving packet, the buffer below the arrival depicts the state of the buffer after handling the packet's arrival. The packet in the queue's head-of-line (HoL) at the end of the arrival step is emphasized by an extra, internal, square. This packet is the one processed in the processing step. The state of the buffer at the end of each cycle is highlighted with light-gray background.