QueuePilot: Reviving Small Buffers With a Learned AQM Policy

Micha Dery, Orr Krupnik, Isaac Keslassy
Technion

***Abstract*—There has been much research effort on using small buffers in backbone routers, to provide lower delays for users and free up capacity for vendors. Unfortunately, with small buffers, the droptail policy has an excessive loss rate, and existing AQM (active queue management) policies can be unreliable.**

We introduce QueuePilot, an RL (reinforcement learning)-based AQM that enables small buffers in backbone routers, trading off high utilization with low loss rate and short delay. QueuePilot automatically tunes the ECN (early congestion notification) marking probability. After training once offline with a variety of settings, QueuePilot produces a single lightweight policy that can be applied online without further learning. We evaluate QueuePilot on real networks with hundreds of TCP connections, and show how its performance in small buffers exceeds that of existing algorithms, and even exceeds their performance with larger buffers.

I. Introduction

Efficiently reducing buffer sizes in Internet routers may be one of the main outstanding open problems in networking [1]–[10]. The networking community has worked on small buffers for nearly 20 years [3], even organizing dedicated workshops [11], albeit with limited impact on real routers. In this paper, we claim that using RL (reinforcement learning) may help solve this longstanding problem, and demonstrate a first step in this direction.

Small buffers. Internet router buffers help deal with congestion and traffic variability. They trade off three key measures: output link utilization, packet loss and packet delay. If they are too small, they may quickly lead to under-utilization when there are no arrivals, and high loss rates upon many arrivals. If they are too large, they may lead to large delays.

In 1990, Van Jacobson [1] noted that to prevent under-utilization, a single long-lived TCP flow going through a router bottleneck link using droptail needs a router buffer size B of at least $B \geq BDP = C \cdot RTT$, i.e., the bandwidth-delay product (BDP) of its capacity C and the propagation round-trip time RTT. This is known as the BDP rule-of-thumb [2].

In 2004, Appenzeller et al. [3] found that under several assumptions, to achieve a high utilization in a similar setting, N long-lived TCP flows would need a smaller buffer size $B \geq BDP/\sqrt{N}$. For example, a 400-Gbps backbone router link carrying 100 K flows with an average RTT of 60 ms would need 9 MB instead of 3 GB, a $\sqrt{N} \approx 300\times$ reduction [12], [13]. In this paper, we define buffers of size $BDP/\sqrt{N}$ as *small buffers*. Smallness is relative: 9 MB allows for some 6,000 Ethernet-sized packets. Similar small-buffer results have since been confirmed for additional congestion control algorithms (Cubic, BBR, etc.) [10] and for real backbone networks (Level 3, Google, Microsoft, etc.) [8], [10]. Additional papers have argued that buffer sizes could be further reduced and that by using *tiny buffers* of about 50-100 packets, a backbone network could still reach over 90% utilization, as long as there is packet pacing at the sender or at the edge [4], [6].

Router vendors would be eager to reduce buffer sizes. The BDP rule of thumb is still the basis for designing current backbone router buffers, albeit with a reduction because of its huge size [9]. For example, Cisco's 400-Gbps linecards offer either 18 ms (Q100 linecard) or 13 ms of buffering (Q200 linecard) [14]. Modern forwarding ASICs can waste half their capacity accessing off-chip memory, while smaller buffers could enable on-chip SRAM buffering, as in datacenter switches [8], [15]. Thus, smaller buffers could translate into cheaper, less power-hungry and faster routers.

Users would also be eager to reduce buffer sizes. Many papers have reported large increases in queueing delays at the core during times of congestion [10]. A buffer-sizing experiment at Netflix found an impact on video performance [7].

Unfortunately, with a droptail policy, small buffers incur high drop rates that can violate SLAs (service-level agreements), preventing operators from implementing them [5], [8]. An ECN (early congestion notification)-based AQM (active queue management) policy such as RED (random early detection) [16] is appealing, as it can mark packets instead of dropping them, and consequently lower the drop rate. However, it also lowers utilization, and more significantly, it is deemed too sensitive to parameter tuning to be implemented in backbone networks [17]–[20].

Automatic AQM. Among the many AQM algorithms [21], three have been standardized as IETF RFCs [22]–[24]: RED [16] which was updated later with its adaptive variant (ARED) [25], CoDel [20], and PIE [17]. ARED was introduced to solve the sensitivity to parameters and traffic load changes. Nonetheless, selecting the target queue size is left to the network operator. In later work, CoDel and PIE were crafted to manage large buffer sizes and offer a "no-knobs" AQM without parameter tuning. Nevertheless, later studies [26], [27] found that ARED provides comparable results to CoDel and PIE, and that their parameters of choice do not achieve the best results and also need to be tuned. AQM can be cast as a sequential decision-making problem, for which RL is a powerful tool. Four recent papers have introduced RL for AQM: QRED [28] learns to adjust the RED thresholds for dropping; DRL-AQM [29] and RL-AQM [30] directly adjust the packet drop probability; and ACC [31] learns to adjust

the RED parameters for ECN marking in datacenter switches. These papers make the case that learning can help obtain better results than existing AQM policies. They also show how learned policies are typically more resilient to changes than existing ones. However, only ACC performs experiments on real networks. Furthermore, none of these papers focuses on the impact of small buffer sizes, which is the focus of this paper.

Contributions. The main contribution of this paper is QueuePilot, an RL-based algorithm for improving performance in a backbone router with small buffers. We design QueuePilot to be fully autonomous and run in a backbone router using local buffer information only, as do current AQM algorithms. We also opt to train a single policy offline in a variety of settings and keep reusing this single policy online in any setting. QueuePilot adopts a general RL-based AQM policy that directly controls ECN marking. Furthermore, we keep the implementation and training process simple, in order to enable a backbone router to run it in real time with no GPU. Finally, we implement QueuePilot using eBPF, leveraging fast kernel access that enables us to use short time-steps of 5 ms.

To evaluate QueuePilot, we run experiments with hundreds of flows on a real testbed. Our experimental results show the following: (1) In a congested router, QueuePilot displays stable performance across a wide range of buffer sizes, outperforming a) droptail, b) ARED with various parameters and c) ACC. (2) The reward performance of QueuePilot at the small-buffer size $B = BDP/\sqrt{N}$ exceeds that of these algorithms at any buffer size. At the small-buffer size, QueuePilot further exhibits strong FCT (flow completion time) tail properties. QueuePilot also displays promising performance at an even smaller buffer size. (3) QueuePilot adapts to varying degrees of congestion, always obtaining the best performance in known environments, and generally obtaining better performance in previously unseen conditions. It also displays some unexpected marking strategies. (4) When training QueuePilot on a single congested router and testing it on a parking-lot topology with two congested routers, we find that it can generalize and obtain good performance. QueuePilot code is available at [32].

II. QUEUEPILOT

In this section, we introduce our QueuePilot algorithm. We present our key design choices (§ II-A), then delve into the details of the RL formulation (§ II-B), and conclude with our RL-agent and eBPF implementations (§ II-C).

A. Key design choices

We begin by presenting the key choices we made in the process of designing the characteristics of QueuePilot.

Autonomous. We designed QueuePilot to be fully autonomous. It runs for a specific router buffer, and only considers information local to this router buffer. While we have considered the benefit of obtaining information from neighboring buffers or neighboring routers, we settle on a simpler solution where QueuePilot can incrementally replace current AQMs.

Offline training. We opt for a train-once-offline-and-reuse-online policy. Unlike ACC, we do not allow online training, for two reasons. First and foremost, we want to avoid the well-known *catastrophic forgetting* phenomenon [33]–[35], whereby new samples cause a learned policy to forget old information and lose some of its generalization properties. Second, a static policy is amenable to better debugging properties.

Single policy. We attempt to train a single policy that will be valid for all buffer sizes and all topology settings. This is a significant decision. As we show in the experiment results in § III-B, such a policy may trade off performance for generality, and perform slightly worse than a policy that is custom-trained on a single scenario, e.g., for a given buffer size. However, adopting a different policy for each setting combination (e.g., buffer size, bandwidth, estimated number of flows, proportion of TCP traffic, proportion of ECN-capable traffic, etc.) could involve an exponential number of policies. A single policy is appealing to operators for its simplicity.

General policy. To keep our policy as general as possible and allow it to take the widest possible set of actions, we do not restrict our action choices to the parameters of RED (which is the approach taken by QRED and ACC). Instead, the actions of our agent directly control the ECN marking probabilities.

ECN marking. We opt to mainly signal congestion using ECN marking, and only drop packets when the buffer is full. This enables a softer state-signaling than dropping packets, especially as we would like to reduce drop rates to fit SLAs. This is made possible by the increased adoption of ECN-capable hosts (above 81% in 2018 [36], probably around 90% today). When attempting to mark a packet which is not ECN-capable (e.g., UDP), we choose not to drop it, which makes it even harder to maintain small queues. This is a well-known fairness issue [37], [38].

Compatibility. QueuePilot should easily fit within the AQM library of router vendors. Therefore, we only rely on simple existing aggregate buffer measures (e.g., queue size or in/out throughput) rather than on per-flow measures. Tracking elephant flows may improve performance, but is outside the scope of this work.

Long history. Since we do not know the exact RTT, QueuePilot should rely on a history of unspecified length. We equip our model with an LSTM (long short-term memory) module [39] that does not take as input a fixed-length history. The standard RL practice of using a discount factor regulates the impact of temporally remote events.

No GPU. We aim for a lightweight policy that a backbone router could run in real-time on real traffic. The policy should be simple enough that it can run reasonably fast on a computer without a GPU.

Low-level access. We implement and train QueuePilot on a network of Linux-based PCs and run it in real-time. We use eBPF, enabling a user program to access the packets in the kernel-space for marking and dropping, for example. Using unoptimized Python, we collect the observations, then compute and apply the policy decisions every 5 ms.

B. Reinforcement learning

We assign the AQM task to an RL agent that controls one egress-port router buffer in a partially-observable environment. We divide time into slotted sequential intervals denoted as *steps*. At each time step, the agent receives statistics about its controlled buffer and produces an action that is held for the rest of that step.

Formulation. In the RL literature, a decision-making problem is described as an MDP (Markov decision process) [40] parameterized by the tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{T}, r, \gamma\}$, where $s \in \mathcal{S}$ describes a *state* of the system, $\{a \in \mathcal{A}\}$ are the *actions* available to the agent and $\mathcal{T}(s_{t+1}|s_t, a_t)$ is the transition probability at time step t from the current state s_t to the next, given the action a_t applied by the agent. In addition, $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function, providing reward r_t to the agent at each time step given the current s_t and a_t. Finally, $\gamma \in (0, 1)$ is a discount factor controlling the effect of temporally remote states. The objective of the agent is to maximize the expected cumulative discounted reward $\mathbb{E}_{s,a} \sum_t \gamma^t r_t(s_t, a_t)$ by learning a policy $\pi : \mathcal{S} \to \mathcal{A}$ governing its behavior. In some cases, the agent cannot directly access the state of the system s_t, but instead views some *observation* $o_t \in \mathcal{O}$ generated from the state at each time step by an emission function $g : \mathcal{S} \to \mathcal{O}$ (e.g., QueuePilot can access the current buffer utilization but not the state of the end-host CCAs). The policy then operates on the observations, and not directly on the states: $a_t \sim \pi(o_t)$. This extension of the RL formulation is known as a *Partially-observable MDP* (POMDP) [41].

Raw statistics. As Fig. 1 illustrates, we monitor the following raw statistics: 1) *number of received (Rx) packets* and 2) *transmitted (Tx) packets*; 3) *number of dropped packets*; 4) *number of packets marked for congestion*; and 5) *queue length*.

Observations. We construct an observation space for the AQM task using the above raw metrics that are available within the router where the agent operates. At each step t, we compute the following observations: 1) *Relative Rx rate* Rx/C, defined as Rx divided by C, the output link capacity in a single time step; 2) *link utilization* $U_t = Tx/C$ ($U_t \in [0, 1]$); 3) *packet drop rate*, defined as the number of dropped packets relative to C; 4) *packet marking rate*, i.e., share of marked packets out of all Rx packets; 5) *average queue length* Q_t, an EWMA (exponentially-weighted moving average) of the queue length; 6) *average queue delay* $D_t = Q_t/C$ for a new incoming packet [17] (excluding transmission time).

We use the average queue length to consolidate the occurrences in the queue within the step with increased importance to recent values. We use normalized or relative observations, which is a common practice to avoid policy dependency on absolute values and to improve generalization.

History. Our agent receives observations sequentially over the course of interaction with the system, and aggregates them using an LSTM module, a type of Recurrent Neural Network module that collects sequential information and maintains an internal calculated state. In this manner, the agent can implicitly store a history of observations, which may depend on previous actions, and infer the underlying system state s_t as part of its decision-making process.

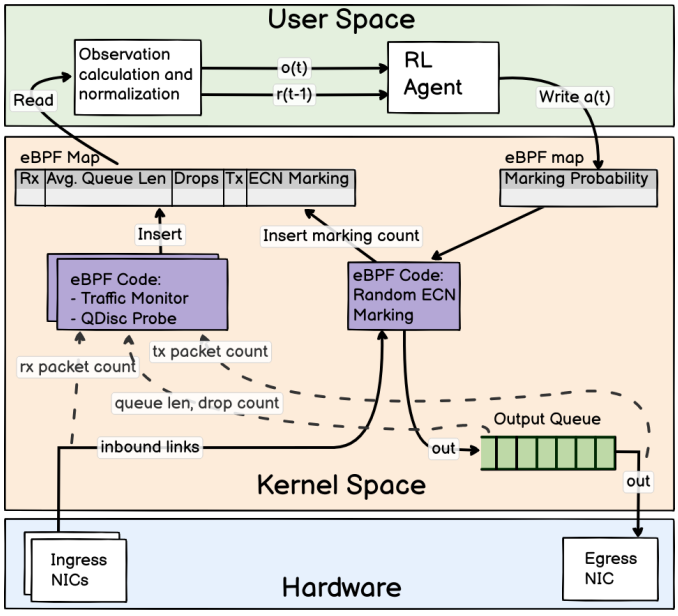


Fig. 1. System overview separating the hardware, kernel-space and user-space layers: an eBPF code (purple, left) collects statistics, organized as an observation vector, and passes them to the RL agent. The agent's action is a marking probability that is applied by the eBPF marking code (right).

Actions. ECN can signal to the sender to adjust its rate without requiring a loss of goodput. Our agent sets the probability for ECN marking, i.e., marking packets with a CE (congestion experienced) bit in the IP header. It does not drop packets at any point other than when the buffer overflows. We use a simple non-uniform quantization of the probability range to construct a small action space: $\mathcal{A} = \{0, 0.01, 0.05, 0.1, 0.5, 1\}$. This action space is sufficient as high probabilities have a drastic effect on the traffic load and induce a drop in throughput by the senders; therefore, we did not find an advantage in including more fine-grained actions between 0.5 and 1. After the agent receives the step observations o_t, it samples an action $a_t \in \mathcal{A}$ that is applied to the current step. Each packet's CE bit is then marked with probability a_t at the enqueue stage. We use random marking to reduce synchronization between different flows. To allow the agent full control of its policy, we do not set restrictions on consecutive marking probabilities and the action set $\mathcal{A}$ is never masked.

Reward. A network operator would define a reward function that reflects a high throughput while maintaining a low queuing delay and low loss rate. As a result, we directly optimize the metrics that are pertinent to our task, notably by using the delay itself and not the queue size. To that end, we define the following reward function:

$$r_t = \begin{cases} -1, & \text{if packet drop at time } t > 0, \\ \frac{{U_t}^2}{\sqrt{1+D_t}}, & \text{otherwise,} \end{cases} \tag{1}$$

using the previously-defined utilization U_t and average delay D_t. This non-linear reward ranges within $[-1, 1]$. Packet drops

are punished (1st line), signaling to the agent that overflow is allowed only if it is worthwhile in an accumulated reward perspective. When there is no packet drop (2nd line), the reward increases with the utilization and decreases as the average delay accumulates. While we focus on small buffer sizes, this reward is suitable for any buffer size.

Since the reward balances different objectives, it requires some refinement. We initially designed the 2nd line of the reward as a linear combination of U_t and D_t. However, we found that a linear design does not perform as well as a non-linear one in extreme conditions. In particular, with low queue delays, it suffers from a lower-magnitude gradient (a point established in [31]). Thus, we considered $\frac{U_t}{1+D_t}$. We then found that the resulting algorithm assigns too much weight to the delay at the expense of utilization. For instance, a utilization of 1 with a delay of 5 ms (one step) yields a step reward of 0.5, equivalent to a utilization of 0.5 with zero delay. Therefore, we give more weight to utilization by squaring it and less to delay by taking its square root.

C. Implementation

The implementation is driven by the key design choices defined in § II-A. We implement and train QueuePilot on a real network testbed, and run it in real-time with hundreds of TCP connections and UDP background flows.

RL agent. As our RL framework, we use the RLlib v1.11 [42], [43] implementation of PPO [44], a widely-used actor-critic algorithm. We integrate this framework with a custom OpenAI Gym [45] environment that interacts with the Linux kernel using eBPF. See Fig. 1 for an overview of our implementation and agent integration. As our agent architecture, we use a shared initial LSTM feature extractor, which receives new observations at each time step and aggregates them in its 32-dimensional hidden state vector. This is followed by two fully-connected neural networks for the actor and critic components, each with two layers of 64 hidden units.

eBPF. QueuePilot runs on Linux-based PCs that form our network. In this setting, packet ECN marking and observation sampling are tasks that require access to kernel space. eBPF is a technology that provides the option to run a user program within the OS kernel space, without the need to recompile the entire kernel. Particularly, we are interested in network event tracepoints and packet processing functions. We use BCC [46] to implement and load our eBPF programs. We construct a set of functions that are attached to tracepoints, probes and filters, as shown in Fig. 1 (dashed lines). For example, queuing events are monitored with tracepoints and probes at dequeue and enqueue events respectively, and a filter is placed to apply ECN random marking following the agent's action. The ECN marking is applied prior to the enqueuing of the packets, similarly to how ARED operates. We use maps to get gathered statistics to the agent and to pass the agent's action to the random marking function.

Step size. QueuePilot uses a step size of 5 ms, which is shorter than the average RTT. A short step allows the policy to better monitor changes in traffic patterns and be more responsive to them. Since we design QueuePilot to operate under various RTTs, it is an added benefit that the step size is uncoupled from any RTT assumptions.

III. Experiments

In this section, we evaluate the performance of QueuePilot with real testbed experiments. We compare QueuePilot to droptail, ARED and ACC. We first evaluate a simple topology with a single congested router; then, we experiment in a more complex parking-lot topology. We seek to answer the following questions regarding QueuePilot:

1) How does it perform across several buffer sizes? How does it trade off the various reward components? (§ III-B)
2) In small buffers, does its better reward translate into a better FCT tail? (§ III-C)
3) In small buffers, how does it perform in both seen and unseen configurations? (§ III-D)
4) In small buffers, does it automatically adapt in real time to varying traffic intensities? (§ III-E)
5) In small buffers, can it generalize to a multi-router topology where AQMs at congested points interact? (§ III-F)

Note that we also implemented QueuePilot in Mininet [47], with the hope of evaluating QueuePilot in extensive topologies. However, several issues convinced us to train and test on the testbed only. First, due to computational limitations, we could at most run a few dozens of TCP connections on Mininet. Second, we found that with our small step size, Mininet queue and flow dynamics greatly differ from those of a real network, probably due to OS resource-allocation issues. Thus, a successfully trained algorithm on Mininet would underperform when evaluated on real computers.

A. Testbed settings

Topology. We begin our testbed experiments with a dumbbell topology having a single congested router. 300 TCP sources are connected to 300 destinations through two intermediate routers R_1 and R_2 that share a congested link. We host all the sources on a single computer, connected to R_1 using a 40-Gbps link rate. Then the congested link rate between R_1 and R_2 is 1 Gbps. R_2 is connected to the destinations hosted on a separate computer using a 1-Gbps link. Link rates are symmetric. The routers are output-queued. Congestion takes place in an egress buffer of R_1, where we run our AQM agent.

Implementation. Our configuration is composed of four Ubuntu 20.04 PCs. They respectively implement the sources, R_1, R_2, and the destinations. The PC for R_1 is equipped with an Intel Core i7–11700K CPU and 32 GB DDR4 SDRAM, without a GPU. Router R_2 is essentially pass-through since its output rate is equal to its input rate. R_2 adopts a droptail discipline. All offload operations to the NICs, e.g., segmentation offloading, are disabled to provide reliable measurements and packet marking of the traffic passing through the congested router. An equal delay is added with a NetEm [48] implementation in the Linux kernel at the egress of the sending and

receiving hosts. For instance, an RTT of 30 ms reflects an added delay of 15 ms at the egress of each end-host. We use iproute2 v5.18 [49] for Linux's tc (traffic control) updates. MTU is left unchanged at 1500 Bytes. ECN is enabled by default for TCP and ECN fallback is disabled as we guarantee ECN support in our setup.

Traffic. We use iperf3 v3.11 [50] to generate TCP and UDP traffic. TCP connections are split equally between CUBIC and New-Reno connections [51]. TCP connections for the FCT (flow completion time) experiment are sent with nuttcp v6.1.2 [52], since iperf3 handles closing connections differently [53], and are measured with TShark v3.6.5 [54]. All tests have (non-ECN capable) UDP background traffic, at a constant rate of 80 Mbps (8% of the 1-Gbps congested link rate).

Algorithms. We compare our results to droptail, ARED and ACC when applicable. With the exception of droptail, we set all algorithms to use ECN to mark packets upon congestion instead of dropping them, as one of our goals is to decrease drop rate. When we train QueuePilot and ACC, training time is the same for both and long enough for the agents to converge. Note that the bottleneck in the training time is in running on a real network, rather than the computation required for updating the agent's policy, which is orders of magnitude shorter.

Droptail. Droptail uses a single-priority pfifo queue [55].

ARED. We use Linux's tc default ARED implementation in the Linux kernel with the ECN option set. We compare at first three ARED variants $\{\text{ARED}_i\}_{i=1,2,3}$, with minimum threshold $min_{th} = 0.1 \cdot i \cdot B$ given buffer size B. Other parameters are chosen to provide empirical best results and set according to the guidelines in [25] whenever possible. Additional eBPF functions are applied to monitor observations under ARED with added probes to retrieve ARED internal variables.

ACC. Due to the lack of an open implementation of ACC, we implement the algorithm as described in [31] based on Linux's tc RED, and complete missing details with the same choices we make in QueuePilot. ACC's training process combines 2 phases: (1) offline and (2) online. The offline training is done with data from various network scenarios, and the subsequent online training takes place in a particular switch and aims to fine-tune its performance to the current scenario. We implement the two ACC policies resulting from both phases. When running experiments over a set of network settings (e.g., with a set of different buffer sizes): (1) $\text{ACC}_{\text{Phase1}}$ is a single policy trained similarly to QueuePilot on the entire set of settings; while (2) ACC is additionally trained each time for the specific setting on which it is tested.

QueuePilot. Like droptail, we implement QueuePilot in a single-priority pfifo queue, and can trigger new actions (e.g., change the marking probability) every 5 ms. QueuePilot uses a single policy that we train on the entire set of settings. In addition, we attempt to better understand the cost of our choice of adopting a single QueuePilot policy rather than training a set of policies. To this end, we introduce QueuePilot*, a version of the previous single-policy agent which we continue to train each time for the specific tested setting.

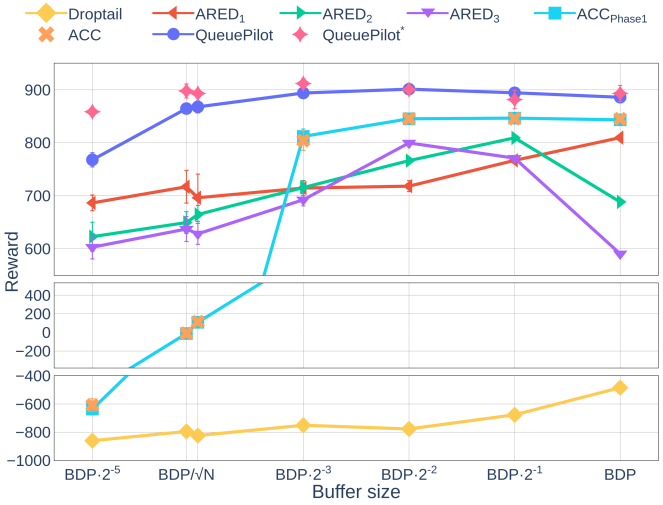


Fig. 2. Single congested router: accumulated reward as a function of buffer size for different algorithms.

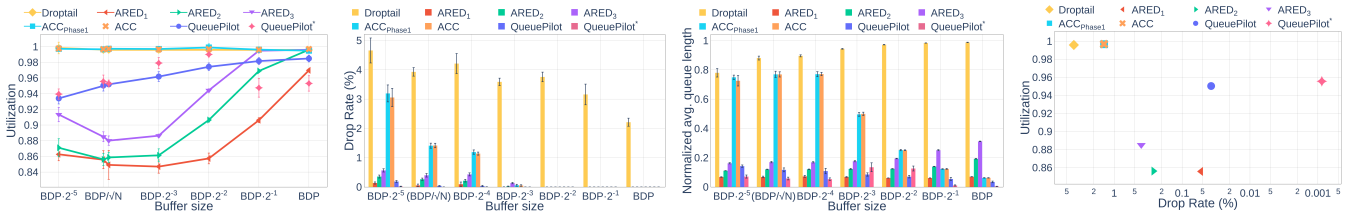
Test process. Unless otherwise stated, we repeat each test 10 times and present the average test results and their standard deviation. In each test, TCP flows start and run for 5 seconds to stabilize before we begin our measurements for 5 additional seconds, or longer when stated. Note that we also ran tests that included the slow-start stage, to similar results.

B. Reward sensitivity to the buffer size

Setup. We start by testing the performance of QueuePilot with different buffer sizes $\left\{BDP \cdot 2^{-i}\right\}_{i\in[0,5]}$, from BDP down to $BDP \cdot 2^{-5}$-sized buffers, while keeping the other network settings unchanged. We run 300 long-lived TCP flows with $RTT = 30$ ms, thus buffer sizes are $B = BDP = 2439$ packets down to $B = BDP \cdot 2^{-5} = 76$ packets, which reaches the *tiny* buffer range of 50-100 packets [4]. The target small-buffer size of $B = BDP/\sqrt{N} = 141$ packets is also added. We train QueuePilot and $\text{ACC}_{\text{Phase1}}$ on the whole range of buffer sizes. QueuePilot* and ACC are additionally trained on each buffer size separately, generating a different policy per buffer size.

Reward. Fig. 2 compares the accumulated reward for different algorithms, given different buffer sizes (higher is better). Each point illustrates the average accumulated reward over all test runs, and its associated bar shows ± one standard deviation (which can often barely be seen). QueuePilot* and ACC are represented without lines connecting the dots to reflect the additional policy training specifically for each buffer size. QueuePilot and QueuePilot* obtain the highest rewards across the range of buffer sizes. QueuePilot's reward slightly decreases with the smallest buffer sizes. We can see that QueuePilot* does not significantly improve over single-policy QueuePilot in most buffer sizes, despite its targeted training.

Focusing on the small-buffer size $B = BDP/\sqrt{N}$, it is significant that QueuePilot's reward is barely worse than its



(a) Utilization (b) Average drop rate (c) Normalized average queue length (d) Util. vs. drop rate at $BDP/\sqrt{N}$

Fig. 3. Single congested router: understanding the reward components. (a-c) present measures that impact the reward, (d) plots utilization vs. drop rate.

own reward for the large buffers, and better than the rewards of any competing algorithm (droptail, ARED, ACC) for any buffer size. Droptail has weak performance across the range. ACC collapses for the small-buffer size. Therefore it does not perform as well as ARED in this setting, despite being an auto-tuning RED AQM. The reward of QueuePilot* is within the margin of error of its reward in the larger buffers (and in fact, slightly higher). Finally, note that for the smallest buffer on the left, QueuePilot* performs well, yielding promising results for learning policies below the small-buffer size.

Deep dive into the reward components. Fig. 3 illustrates three measures that impact the reward function: the link utilization in Fig. 3(a), the drop rate in Fig. 3(b), and the normalized average queue length, i.e., Q/B where Q is the average queue size, in Fig. 3(c). We plot this last component instead of the average expected delay $D = Q/C$ to better understand how each algorithm uses the available buffer resource. Note that B varies, so doubling B will double the delay given the same normalized queue size.

For each buffer size, there is a tradeoff between these three components, and we can better understand the decisions of each algorithm. QueuePilot attempts to obtain a reasonable value in all three components, but is not necessarily the best in each. In particular, in the small-buffer size $B = BDP/\sqrt{N}$, it only obtains a throughput of 95%, which is reasonably good, but below the great throughput of ACC and droptail. On the other hand, its loss rate is under 0.04%, well below ACC and droptail, which exceed 1%. Its average queue size (and therefore average delay) is also significantly lower, although this is less significant in a small buffer. The three components also help us understand why in Fig. 2, different ARED parameters are optimal for different buffer sizes in a non-monotonous way, displaying how ARED is hard to tune: e.g., aggressive ARED_1 (red line) performs better at the large BDP buffer because of its lower delay; worse in middle-sized buffers because the other variants reduce their delays; and best for smaller buffers because the drop rates of other variants increase. Finally, it is interesting that for the large buffers $B = BDP \cdot 2^{-1}$ and $B = BDP$, QueuePilot* converges to a policy that is quite different from QueuePilot, even though the resulting total reward is close (in Fig. 2): it sacrifices its utilization in exchange for a lower queue length.

Pareto efficiency. Fig. 3(d) visualizes the tradeoff between utilization and drop rate for the small-buffer $B = BDP/\sqrt{N}$. It shows how QueuePilot lies on the Pareto-efficient frontier vs. droptail, ARED and ACC. While QueuePilot has a lower utilization than droptail and ACC, it compensates by attaining a particularly low loss rate. As expected, QueuePilot* reaches an even better point, since it optimizes for this buffer size only.

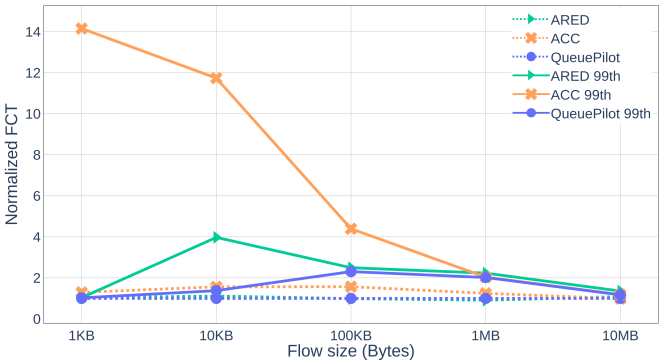
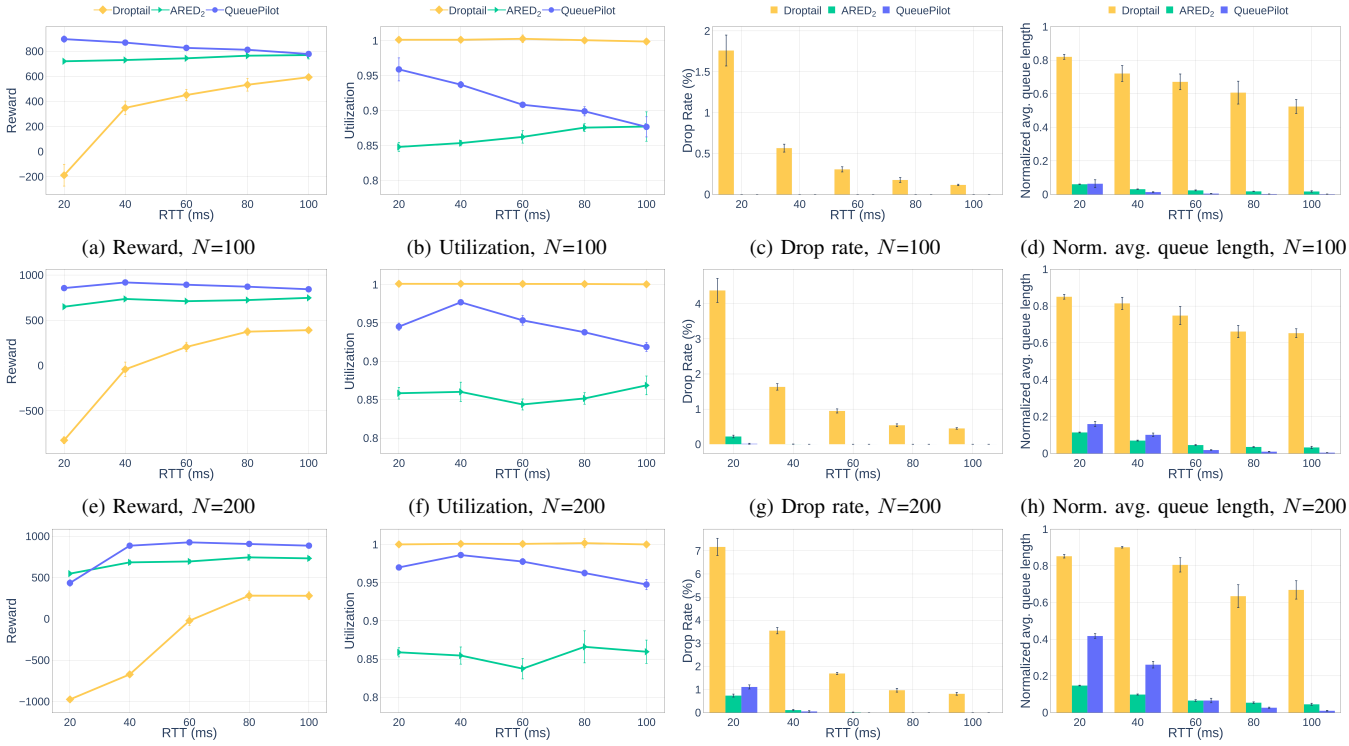


Fig. 4. Mean and 99th-percentile FCT for different flow sizes

C. FCT tail

Setup. We seek to understand the effect of the QueuePilot policy on the end-to-end flow performance in small buffers. To do so, we borrow the FCT (flow completion time) benchmark from [31] and modify it to run with TCP instead of RDMA, achieving near-100% load in our tests. We keep the same network settings as previously used, and focus on the small-buffer size $B = BDP/\sqrt{N}$. We then measure the FCT of TCP flows carrying messages of size $\{1\text{KB}, 10\text{KB}, 100\text{KB}, 1\text{MB}, 10\text{MB}\}$. We start 300 TCP connections with 80-Mbps UDP background traffic and then measure the FCT of 50 additional TCP flows of the same message size. Overall, by repeating the test, we measure 1000 flows for each message size with ARED, ACC and QueuePilot. The QueuePilot agent is the same agent from the previous experiment (§ III-B) trained on the whole range of buffer sizes, while ACC is trained specifically for this buffer size.

Results. Fig. 4 shows the mean and 99th-percentile FCT of each algorithm, normalized by the mean FCT of QueuePilot. For the mean FCT, QueuePilot achieves comparable results with ARED, even slightly worse at times, while ACC obtains 50% longer mean FCT in small messages and comparable results in large messages. The tail of the FCT distribution, as shown by the 99th percentile, shows that in small messages (1 KB and 10 KB) QueuePilot significantly outperforms ARED and ACC, which suffer from up to $4\times$ and $14\times$



(a) Reward, N=100 (b) Utilization, N=100 (c) Drop rate, N=100 (d) Norm. avg. queue length, N=100

(e) Reward, N=200 (f) Utilization, N=200 (g) Drop rate, N=200 (h) Norm. avg. queue length, N=200

(i) Reward, N=300 (j) Utilization, N=300 (k) Drop rate, N=300 (l) Norm. avg. queue length, N=300

Fig. 5. Single QueuePilot policy performance with (a-d) 100, (e-h) 200 and (i-l) 300 TCP flows in different RTT scenarios

normalized FCTs respectively, vs. up to $1.4\times$ for QueuePilot. With large messages, ARED's tail is the longest with 33% longer FCT for 10 MB messages.

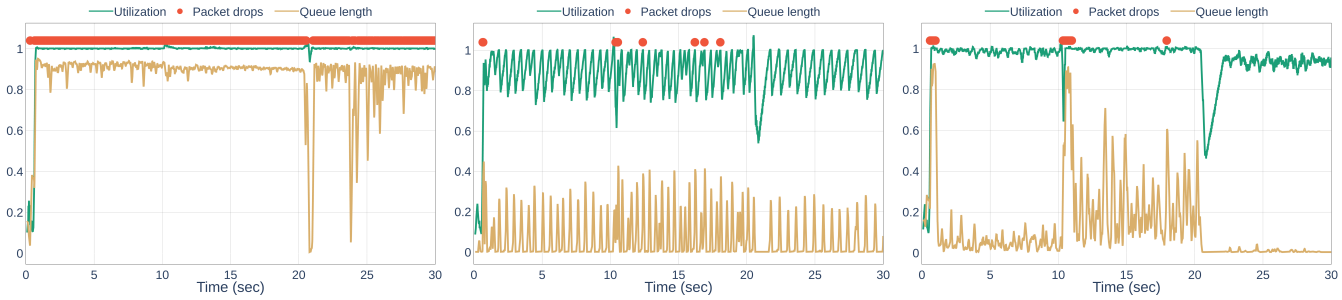
This is another tradeoff example. High drop rates can be extremely costly to the user when the message size is small, but throughput takes the upper hand in importance for a large message size. Note that the FCT achieved by a TCP flow correlates to all 3 metrics: link utilization, drop rate and delay. In that sense, QueuePilot offers a suitable balanced solution with the $BDP/\sqrt{N}$ buffer size, with extremely low drop rate and relatively high link utilization.

D. QueuePilot sensitivity to the environment

Setup. We argue that RL has the capability to learn various traffic patterns and to react to them properly with a single policy. In this experiment we show that QueuePilot can steer small buffers through various traffic environments. We first train a single QueuePilot agent in our dumbbell topology while constantly changing the training environment: At each training run, we randomly select a number N of TCP flows within $\{100, 200, 250\}$, and RTT within $\{20, 40, 60, 80\}$ ms. We update the buffer size $B = BDP/\sqrt{N}$ accordingly. When testing, we vary the number N of TCP flows within $\{100, 200, 300\}$, keeping in mind that $N = 300$ falls outside the training range of QueuePilot. We also vary RTT within $\{20, 40, 60, 80, 100\}$ ms, noting again that 100 ms falls outside the training range of QueuePilot. We update the buffer size $B = BDP/\sqrt{N}$ according to each chosen setting.

In this and the following experiments, we compare QueuePilot to droptail and ARED only, since droptail is the current default solution, and since in small buffers ACC reward collapses vs. ARED as ACC becomes a less successful adaptive RED AQM. In addition, for readability, the following figures contain only one parameter set for ARED. We find that other parameters lead to similar results.

Results. Fig. 5 plots the experiment results and their standard deviation. Each line corresponds to a different number of TCP flows. Each column presents a different metric: reward, utilization, drop rate, and normalized average queue length. The x-axis represents the RTT. As shown in Figs. 5(a), 5(e), and 5(i), QueuePilot always achieves a higher reward than ARED and droptail across the different environments on which it was trained, and generally achieves a higher reward in the new environments. We find that QueuePilot is weak in two corner cases, both outside its training range, corresponding to the smallest and largest buffer sizes in the testing set: (1) Its reward is worse than ARED in the setting with $N = 300$ TCP flows and $RTT = 20$ ms (Fig. 5(i)). Since the buffer size is $B = BDP/\sqrt{N}$, a low RTT and a large N corresponds to the smallest buffer size in the testing set ($B = 94$ packets, within the tiny range). QueuePilot has trouble adapting to the increased traffic burstiness and yields a higher drop rate than ARED. (2) Its reward is still the best one with $N = 100$ TCP flows and $RTT = 100$ ms (Fig. 5(a)), but like QueuePilot* in Fig. 3, it sacrifices its utilization in exchange for a lower queue length. This may not intuitively be the preferred behavior. Note



(a) Droptail (b) ARED (c) QueuePilot

Fig. 6. Dynamic traffic intensity with a small buffer: packet drops, utilization and average queue length with 200 flows at $t=0$, 300 at $t=10$ sec and 150 at $t=20$ sec.

that we keep using the small-buffer size formula throughout this experiment; therefore, queue length and delay are less significant as they are low anyway.

To sum up, in this experiment we find that QueuePilot always obtains the best reward performance on known environments, and generally, but not always, better reward performance with unseen conditions.

E. Automatic tuning

Setup. In this experiment we analyze how QueuePilot can adapt in real-time to changing traffic patterns while maintaining high utilization and low drop rate. We continue using the same trained QueuePilot policy from the previous experiments. We set $RTT = 40$ ms on our dumbbell topology. At time $t = 0$, we begin sending $N = 200$ TCP flows through the network. At $t = 10$ sec, we increase N to 300 flows. Then at $t = 20$ sec, we drop N to 150. UDP background traffic is set to 80 Mbps. Buffer size is set to our target small-size $B = BDP/\sqrt{N} = 230$ packets, where $N = 200$, i.e., it does not change throughout the experiment. We compare QueuePilot to droptail and ARED.

Results. Fig. 6 plots the utilization and queue length changes over time for droptail, ARED and QueuePilot. The two metrics are averaged over 100-ms windows. We also use a red circle to mark when the buffer overflows and packets are dropped. Recall that we show the average queue, while packet drops are caused when the instantaneous queue overflows the buffer.

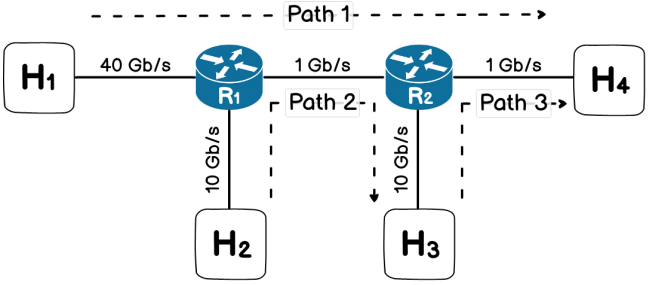
As expected, droptail keeps a near-constant utilization at 1, but its average queue length and drop rate are extremely high. ARED does not seem to change its policy when the traffic intensity changes. It is also near-periodically causing under-utilization of the link in all traffic load options, because the instantaneous queue length can grow fast resulting in ARED's aggressive marking and the flows backing down quickly. QueuePilot achieves almost full utilization in the first two traffic patterns with 200 and 300 TCP flows respectively, but lower utilization in the last pattern with 150 flows ($[20-30]$ sec). Nevertheless, its average utilization is always higher than ARED's.

We note an interesting marking strategy of QueuePilot: It repeatedly uses small marking probabilities, but both their frequency and selected marking probability increase with the queue size and with the load. Interestingly, even with low queue sizes and low loads, QueuePilot still marks packets at times even though signs of developing congestion are barely distinguishable. However, the policy completely stops marking when utilization falls, e.g., in the $[20, 22]$-sec interval.

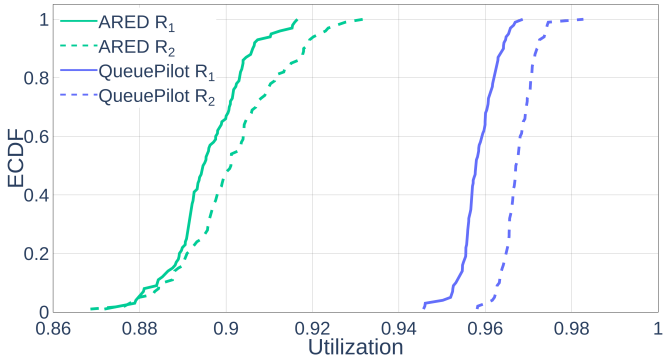
Note that some utilization values are greater than 1 in our results. In our implementation we sample all observations every time step of 5 ms. Since we use unoptimized Python running as a user-space program, the sampling may occasionally be triggered in periods that are slightly longer than 5 ms, thus leading to values larger than 1.

F. Parking-lot topology

Setup. We evaluate QueuePilot in a testbed with two congested routers, after training it with a single congested router, in order to test its topology generalization properties. As Fig. 7(a) illustrates, we form a parking-lot topology through two intermediate routers R_1 and R_2. Two computers, H_1 and H_2, are connected to R_1 via 40-Gbps and 10-Gbps link rates respectively. H_3 and H_4 are connected to R_2 with 10-Gbps and 1-Gbps link rates respectively. The connection between R_1 and R_2 is 1 Gbps. We send traffic along three paths: (1) from H_1 to H_4; (2) from H_2 to H_3; and (3) from H_3 to H_4. In this pattern, congestion is formed in output-queued buffers of both routers: in R_1 before the link to R_2 and in R_2 before the link to H_4. TCP flows in path (1) experience the congestion in both R_1 and R_2, while flows in (2) or (3) experience congestion in R_1 or R_2 respectively. Lastly, a 20-ms delay is set at the egress link of each source or destination, thus $RTT = 40$ ms. We send 150 TCP flows along path (1), 100 along (2) and 100 along (3). We evaluate performance at the buffers of the congested 1-Gbps links in R_1 and R_2. The buffer size in both routers is set to $B = BDP/\sqrt{N} = 206$ packets, as $RTT = 40$ ms and $N = 250$ in each buffer. Each path also contains 50-Mbps background UDP traffic, thus totaling 100 Mbps for every congested buffer, i.e., 10% of the link capacity. We run 100 tests with QueuePilot on R_1 and R_2, and repeat for ARED on both routers. We omit droptail, since we know that it has good utilization but prohibitive loss rate. We continue using the same policy for QueuePilot that was trained with a single congested router (§ III-D).



(a) Two-router parking-lot topology



(b) Utilization distribution in two-router topology with small buffers

Fig. 7. Two-router (a) topology and (b) utilization CDF

Results. Fig. 7(b) shows the empirical CDF of the average utilization in each router for QueuePilot and ARED. As shown, despite the complex two-router interactions, QueuePilot manages to maintain relatively high utilization in both routers independently, along with relatively low standard deviation. This augurs favorably for its ability to generalize to unseen and more complex topologies.

Note that QueuePilot and ARED also achieve low drop rates: for QueuePilot, $7.5 \cdot 10^{-7}$ in R_1 and $1.35 \cdot 10^{-5}$ in R_2, and for ARED, $8.78 \cdot 10^{-6}$ in R_1 and $5.1 \cdot 10^{-5}$ in R_2. Likewise they achieve low normalized average queue lengths: for QueuePilot, 0.117 in R_1 and 0.11 in R_2, and for ARED, 0.085 in R_1 and 0.09 in R_2. Due to the small-buffer size, queue size and delay are low anyway and therefore less significant.

IV. Discussion: Limitations and Future Work

Additional settings. Clearly, it is hard to convince operators to adopt a new AQM policy. There is an endless list of additional settings that should be tried in future work, beyond those that could be presented in this limited space: e.g., consider more complex topologies; consider more complex traffic patterns, including a wide range of mixed RTTs; consider more CCAs (congestion control algorithms), including ECN-insensitive CCAs, RL-generated CCAs, and BBRv2 (as in [10] and Fig. 16 of [56]); introduce TCP short flows; mix different AQMs on different routers; etc.

More metrics. Beyond the FCT presented in this paper, we could consider the impact of backbone queueing properties on traffic from specific applications.

Fairness. The current reward does not consider fairness among flows. A more complex reward could introduce fairness, at the cost of increased implementation complexity.

Guarantees. As a model-free deep-RL solution, QueuePilot is entirely devoid of performance guarantees.

V. Related Work

The introduction (§ I) has provided an extended overview of buffer-sizing and AQM related work. We expand here on remaining topics:

Learning the buffer size. Given an existing AQM policy, ABS [57] learns to tune the buffer size online. It is also possible to do so without learning [58], [59]. This approach is complementary to QueuePilot: e.g., it could use QueuePilot as an AQM policy.

CCA for short buffers. CCAs can be designed to perform well under short buffers. Flashpass [60] relies on a proactive sender-driven CCA to withstand small buffers. Moreover, additional approaches rely on a distributed joint CCA-router approach. MACC [61] suggests a clean-slate policy that introduces learning in both the AQM and CCA and trains all agents in a multi-agent RL system. It may be more relevant to private datacenters than to the internet. Moreover, PERC [62] assumes routers can tell flows about their preferred rates. In contrast, QueuePilot assumes existing CCAs and only manages the AQM policy.

Buffer management. ABM [63] proposes to extend the AQM policy by combining it with the buffer management scheme. QueuePilot could be extended in the same manner.

VI. Conclusion

In this paper we introduced QueuePilot, an RL-based AQM for backbone routers that trades off high utilization with low loss rate and short delay. In our testbed experiments, we showed how QueuePilot obtains consistently high performance across a large range of buffer sizes, despite being a single lightweight policy that is applied online in each setting without further learning. Finally, we showed promising results for QueuePilot in a multi-router topology, suggesting appealing lines of future work for generalizing QueuePilot.

In future work, we hope to investigate whether a learned policy could provide some limited guarantees given a simplified TCP model.

We have provided public access to the QueuePilot code [32].

Acknowledgment

The authors would like to thank Roy Mitrany for his crucial help with the testbed experiments, and Kfir Toledo, Tsahi Birk and the anonymous reviewers for their valuable feedback.

This work was partly supported by the Louis and Miriam Benjamin Chair in Computer-Communication Networks, the Israel Science Foundation (grant No. 1119/19), Toga Networks, and the Hasso Plattner Institute Research School.

REFERENCES

[1] V. Jacobson, "Modified TCP congestion avoidance algorithm," *Email to the end2end-interest mailing list*, 1990. [Online]. Available: https://www.cs.rice.edu/~eugeneng/teaching/f06/comp529/papers/vanj.90apr30.txt

[2] C. Villamizar and C. Song, "High performance TCP in ANSNET," *ACM SIGCOMM CCR*, vol. 24, no. 5, pp. 45–60, Oct. 1994.

[3] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," *ACM SIGCOMM CCR*, vol. 34, no. 4, pp. 281–292, 2004.

[4] M. Enachescu *et al.*, "Part III: Routers with very small buffers," *ACM SIGCOMM CCR*, vol. 35, no. 3, pp. 83–90, Jul. 2005.

[5] M. Shifrin and I. Keslassy, "Small-buffer networks," *Computer Networks*, vol. 53, no. 14, pp. 2552–2565, 2009.

[6] Y. Cai, *Queue length based pacing of internet traffic (PhD thesis)*. University of Massachusetts Amherst, 2011.

[7] B. Spang, B. Walsh, T.-Y. Huang, T. Rusnock, J. Lawrence, and N. McKeown, "Buffer sizing and video QoE measurements at Netflix," *Workshop on Buffer Sizing*, 2019.

[8] N. McKeown, G. Appenzeller, and I. Keslassy, "Sizing router buffers (redux)," *ACM SIGCOMM CCR*, 2019.

[9] L. Wigley, "An update on router buffering," *Cisco White Paper*, Mar. 2022. [Online]. Available: https://xrdocs.io/8000/Buffering-WP_March_2022.pdf

[10] B. Spang, S. Arslan, and N. McKeown, "Updating the theory of buffer sizing," *Performance Evaluation*, 2021.

[11] N. McKeown and C. Diot. (2019, December) Workshop on buffer sizing. Stanford. [Online]. Available: http://buffer-workshop.stanford.edu/program/

[12] M. Trevisan *et al.*, "Five years at the edge: Watching internet from the ISP network," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 561–574, 2020.

[13] Y. Zheng, X. Chen, M. Braverman, and J. Rexford, "Unbiased delay measurement in the data plane," *APOCS*, 2022.

[14] L. Wigley, "8100 & 8200 deployment note," *Cisco XRDocs*, Mar. 2022. [Online]. Available: https://xrdocs.io/8000/blogs/8100-8200-deployment-note/

[15] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson, "Backpressure flow control," *Usenix NSDI*, 2022.

[16] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM ToN*, vol. 1, no. 4, pp. 397–413, 1993.

[17] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg, "PIE: A lightweight control scheme to address the bufferbloat problem," *IEEE HPSR*, 2013.

[18] M. May, J. Bolot, C. Diot, and B. Lyles, "Reasons not to deploy RED," *IEEE/ACM IWQoS*, 1999.

[19] V. Misra, W.-B. Gong, and D. Towsley, "Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED," *ACM SIGCOMM*, 2000.

[20] K. Nichols and V. Jacobson, "Controlling queue delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42–50, 2012.

[21] R. Adams, "Active queue management: A survey," *IEEE communications surveys & tutorials*, vol. 15, no. 3, pp. 1425–1476, 2012.

[22] R. Pan *et al.*, "Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem," RFC 8033, 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8033

[23] K. Nichols *et al.*, "Controlled Delay Active Queue Management," RFC 8289, 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8289

[24] B. Braden *et al.*, "Recommendations on Queue Management and Congestion Avoidance in the Internet," RFC 2309, 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2309

[25] S. Floyd, R. Gummadi, and S. Shenker, "Adaptive RED: An algorithm for increasing the robustness of RED's active queue management," *ICSI tech. report*, 2001.

[26] N. Kuhn *et al.*, "Revisiting old friends: is CoDel really achieving what RED cannot?" *ACM SIGCOMM workshop on capacity sharing*, 2014.

[27] N. Khademi *et al.*, "The new AQM kids on the block: An experimental evaluation of CoDel and PIE," in *IEEE Infocom workshop*, 2014, pp. 85–90.

[28] Y. Su *et al.*, "QRED: A Q-learning-based active queue management scheme," *JIT*, vol. 19, no. 4, pp. 1169–1178, 2018.

[29] H. Ma *et al.*, "An intelligent scheme for congestion control: When active queue management meets deep reinforcement learning," *Computer Networks*, vol. 200, p. 108515, 2021.

[30] D. A. AlWahab, G. Gombos, and S. Laki, "On a deep Q-network-based approach for active queue management," *IEEE EuCNC*, 2021.

[31] S. Yan, X. Wang, X. Zheng, Y. Xia, D. Liu, and W. Deng, "ACC: Automatic ECN tuning for high-speed datacenter networks," *ACM SIGCOMM*, 2021.

[32] (2023) QueuePilot code. [Online]. Available: https://github.com/2dm/QueuePilot

[33] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in *Psychology of learning and motivation*. Elsevier, 1989, vol. 24, pp. 109–165.

[34] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," *PNAS*, vol. 114, no. 13, pp. 3521–3526, 2017.

[35] R. Kemker *et al.*, "Measuring catastrophic forgetting in neural networks," *AAAI*, vol. 32, no. 1, 2018.

[36] C.-X. Chen and K. Nagaoka, "Analysis of the state of ECN on the internet," *IEICE Trans. Inf.& Syst.*, no. 5, pp. 910–919, 2019.

[37] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy, "Virtualized congestion control," *ACM SIGCOMM*, 2016.

[38] G. Judd, "Attaining the promise and avoiding the pitfalls of TCP in the datacenter," *Usenix NSDI*, pp. 145–157, 2015.

[39] I. Goodfellow *et al.*, *Deep learning*. MIT press, 2016.

[40] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[41] L. P. Kaelbling *et al.*, "Planning and acting in partially observable stochastic domains," *Artificial intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[42] E. Liang *et al.*, "RLlib: Abstractions for distributed reinforcement learning," *ICML*, 2018.

[43] (2023) Ray project. [Online]. Available: https://github.com/ray-project/ray

[44] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv:1707.06347*, 2017.

[45] G. Brockman *et al.*, "OpenAI Gym," *arXiv:1606.01540*, 2016.

[46] (2023) BCC. [Online]. Available: https://github.com/iovisor/bcc

[47] K. Kaur, J. Singh, and N. S. Ghumman, "Mininet as software defined networking testing platform," *ICCCS*, 2014.

[48] S. Hemminger *et al.*, "Network emulation with NetEm," in *Linux conf au*, 2005.

[49] (2023) iproute. [Online]. Available: https://github.com/shemminger/iproute2

[50] (2023) iperf3 - A TCP, UDP, and SCTP network bandwidth measurement tool. [Online]. Available: https://github.com/esnet/iperf

[51] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.

[52] (2023) nuttcp. [Online]. Available: https://www.nuttcp.net/

[53] iperf3 issue 382. [Online]. Available: https://github.com/esnet/iperf/issues/382

[54] G. Combs *et al.*, "Tshark-the wireshark network analyser," 2017. [Online]. Available: https://www.wireshark.org

[55] Linux man page. (2023) tc-pfifo. [Online]. Available: https://linux.die.net/man/8/tc-pfifo

[56] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: A pragmatic learning-based congestion control for the internet," *ACM SIGCOMM*, 2020.

[57] J. Tang *et al.*, "ABS: Adaptive buffer sizing via augmented programmability with machine learning," *IEEE Infocom*, 2022.

[58] M. Bachl, J. Fabini, and T. Zseby, "Cocoa: Congestion control aware queuing," *Workshop on Buffer Sizing*, 2019.

[59] E. Kfoury, J. Crichigno, E. Bou-Harb, and G. Srivastava, "Dynamic router's buffer sizing using passive measurements and p4 programmable switches," *IEEE Globecom*, 2021.

[60] G. Zeng *et al.*, "Flashpass: Proactive congestion control for shallow-buffered wan," in *IEEE ICNP*, 2021, pp. 1–12.

[61] J. Bai *et al.*, "MACC: Cross-layer multi-agent congestion control with deep reinforcement learning," *arXiv:2206.01972*, 2022.

[62] L. Jose, S. Ibanez, M. Alizadeh, and N. McKeown, "A distributed algorithm to calculate max-min fair rates without per-flow state," *POMACS*, vol. 3, no. 2, pp. 1–42, 2019.

[63] V. Addanki, M. Apostolaki, M. Ghobadi, S. Schmid, and L. Vanbever, "ABM: Active buffer management in datacenters," *ACM SIGCOMM*, 2022.