Dragonfly: In-Flight CCA Identification

Dean Carmel and Isaac Keslassy

***Abstract*—We introduce the Dragonfly system, which is designed to classify on the fly the congestion control algorithm of any flow that crosses a given router, starting at any time, and quickly reach a reasonable accuracy. To do so, we discuss the unique challenges of real-time congestion control classification. We explain how the number of bytes of the flow within the shared router queue contains an intrinsic memory that significantly helps real-time classification. However, we show that this number of bytes is not straightforward to compute in real time, and introduce ways to do so. We further design an eBPF-based scalable traffic-collection system that helps dynamically filter specific flows at high rates. Finally, we evaluate our Dragonfly system using a variety of platforms, and show that it clearly outperforms state-of-the-art algorithms.**

***Index Terms*—Routers, buffer management, wide-area networks, network protocols, machine learning**

I. Introduction

Cloud providers and internet ISPs face two significant challenges when their routers experience congestion: Determining *why* there is congestion, and finding *how* to handle this congestion. These challenges are especially acute now for cloud providers, who face a convergence of several trends. First, cloud users pay for good network performance and expect it, given the fierce competition between providers [1]. Second, several available VM (virtual machine) tools enable users to quickly pinpoint network issues and blame their cloud providers. Third, cloud users can relatively easily change the CCA (congestion control algorithm) that runs in their VMs. Therefore, they can implement aggressive CCAs that do not provide fairness against competing vanilla TCP flows. For example, Cubic can take up to 80% of link bandwidth when competing with (New)Reno, and BBR can starve competing Reno and CUBIC flows [2]–[9]. Fourth, cloud-based TCP-split proxies can make these CCAs even more aggressive [10]–[12]. Finally, CCAs increasingly use a wide diversity of congestion-detection mechanisms: from loss to delay, from ECN to INT (in-band network telemetry), from destination-based credits to switch feedback, as well as hybrid mechanisms that rely on a subset of these [7]–[9], [13]–[21].

Why? Upon congestion, providers cannot tell *why* it happens. In fact, they cannot even tell *what* CCAs are going through their networks. For instance, when some flows take over the buffers in their routers, providers can know their flow 5-tuples, but no existing algorithm enables providers to easily zoom in on these flows and determine in real time that these are BBR flows that are overpowering their CUBIC traffic. One approach for cloud providers would be to peek at the tenant VMs that cause congestion. But providers do not have easy access to their tenant VMs, not to mention many privacy, security and encryption issues that force them to treat VMs as black boxes [22]. Instead, *in the paper, we focus on the alternative goal of identifying the flow CCA on the fly and locally within the congested router link.* Most existing CCA classification algorithms fail at achieving such a goal, because they often assume that they can also monitor the reverse-path packets, and later provide an offline identification that leverages the packets from both paths [23]–[30]. Unfortunately, this key assumption conflicts with the goal of identifying the CCA *locally* within the congested router link, since the suggested algorithms do not work without monitoring the ACKs on the reverse path. In contrast, DeePCCI [31] does not need to monitor the reverse path, but due to its architecture, it is unclear whether it can easily start monitoring a flow in the middle of transmission, and whether it can scale to more than a handful of flows. Furthermore, Seiðr [32] is designed to run at high line rates, but has trouble handling more than two TCP flavours.

How? Providers also do not have simple tools for *how* to handle congestion. A naive approach would be to apply one of the many fair-queueing [33]–[35] and/or admission-control related algorithms [1], [6], [36], [37]. However, as recently explained by Cebinae [17], fair-queueing algorithms cannot meet the hardware requirements of routers in the internet and public clouds, and admission-control algorithms that drop overflow packets poorly affect non-loss-based CCAs. Instead of using a full-fledged fair-queueing algorithm, Cebinae attempts to achieve reasonable fairness by slightly reducing the rates of a few heavy hitters. However, since Cebinae does not know the CCA of the heavy hitters, it always knocks them with a triple combination of losses, latency, and ECN bits. Heavy hitting the heavy hitters in this CCA-oblivious way may yield large oscillations and potential starvation, again hurting user-perceived performance. Thus, *we are back to the fundamental problem of not knowing the CCA*. If providers could identify the heavy-hitter CCAs, they would fine-tune their congestion signal to each heavy-hitter flow based on its CCA mechanism. For instance, as in P4air [38], they could group all flows with similar CCAs into distinct buffers. Thus, a CCA-aware mechanism would enable providers to handle the large CCA heterogeneity more efficiently. Even with a few mis-classified CCAs, it would be better than the current CCA-oblivious approach. In addition, with tools such as SDN and traffic engineering [39], providers could also reroute hurtful flows and reduce their interactions with regular flows.

Objective. The goal of this paper is to develop a system that can identify the flow CCA on the fly and locally within the congested router link.

Contributions. We introduce Dragonfly, a generic CCA clas-

D. Carmel and I. Keslassy are with the Department of Electrical and Computer Engineering, Technion, Israel (e-mails: deancarmel95@gmail.com, isaac@technion.ac.il).

sification system that enables providers to focus at any time on any flow that crosses their router, and obtain its CCA with a reasonable accuracy. To do so, we introduce *CBIQ* (connection bytes in queue), which measures the number of bytes used by the monitored flow within the router buffer shared by all flows. We explain why CBIQ has unique intrinsic-memory properties that enable it to outperform other measures. We also develop an $O(1)$ algorithm to efficiently compute CBIQ. Finally, we develop Dragonfly around CBIQ using a CNN architecture, and implement a Linux-based eBPF Dragonfly architecture to dynamically select the monitored flows at high line-rates.

We evaluate the Dragonfly system using three platforms: (1) a Mininet network emulation, (2) a physical-network testbed, and (3) a testbed with remote cloud destinations. We show that on all platforms, Dragonfly outperforms state-of-the-art architectures. For instance, when flows are destined to remote cloud servers, Dragonfly achieves an F_1 accuracy score of 0.58 when checking a random sub-interval of 100 ms, and 0.85 with a 1-minute interval. DeePCCI [31] achieves 0.2 and 0.62 accordingly. Finally, in an additional speed experiment, we show how a bespoke sampling method enables eBPF to collect and analyze 20-Gbps traffic.

To summarize, the main contributions of this paper consist in (1) defining the need for on-the-fly CCA classification, (2) introducing and using CBIQ for CCA classification, (3) defining a new algorithm for efficient CBIQ computation, (4) implementing Dragonfly in eBPF, and (5) using first-packet-sampling for eBPF speed acceleration.

The full Dragonfly code is available online [40].

II. Design Goals

We consider a FIFO router buffer that is shared by many flows. We focus on an arbitrary flow, starting at an arbitrary time, and try to determine its CCA.

Minimal router monitor. Our goal is to design a passive CCA classifier that relies on a minimal amount of non-intrusive information from the router. Specifically, *we only intend to monitor the ingress and egress lines*. We are allowed to know when packets of a given flow arrive at the router or depart from it, and use their header information. We should not use information from the ACKs (since they may be in another linecard, if not in another router). For a scalable approach, we should not monitor nor look inside the shared queues, and neither should we use dropping information from the queues.

Real-time. The Dragonfly system should be able to classify any flow *on the fly*, i.e., starting from any point in time. Unlike offline systems that need to collect full information about a flow from start to finish before running a classification process, we want to be able to classify a flow even through we are missing both past and future information.

Scalable traffic collection. The Dragonfly system should be able to scale its traffic collection to high line rates. Ideally, its amount of collected data and its input computation complexity should be in $O(1)$ for each slot, and not scale with the flow rate. It should also be able to scale to a large number of flows, and to dynamically change the list of monitored flows.

Scalable classification. The Dragonfly classifier part should be able to scale to high line rates. To do so, it should rely on a simple neural-network architecture, avoiding expensive components that take significant processing power.

Reliable classification. The Dragonfly system should be able to classify any in-flight flow, starting at any time, and reach a reasonable accuracy in a short time. It should be resilient to the influence of topology settings, of the classification starting time, of additional flows in the shared router queue, and of sampling only some of the packets. Moreover, it should be generic and dynamically adapt to the properties of each congestion control, without requiring bespoke modifications or any detailed domain knowledge of each congestion control.

TCP. In this paper, we focus on TCP CCA variants with available sequence numbers, since they are the most common. In §V, we further discuss this assumption.

III. Dragonfly Algorithm

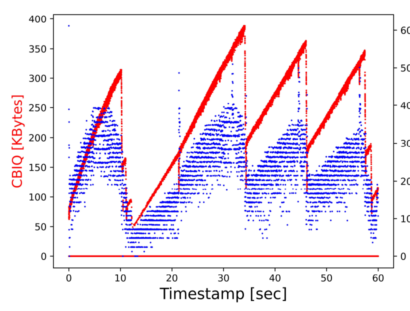
Overview. We use a standard classification framework: given an arbitrary in-flight flow, we start to collect at an arbitrary time a sequence of measures for this flow. This sequence of measures then forms an input array for a neural network, which outputs a guess for the flow CCA. Below, we start by discussing the input measures that should be used for our classification (§III-A, §III-B), then tackle problem-specific traffic-collection (§III-C) and input-computation challenges (§III-D), before describing the neural network (§III-E).

A. Input Aggregation

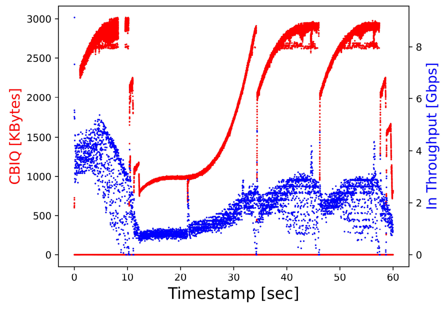
Per-packet raw inputs. We want Dragonfly to classify the CCA of a monitored flow based on information that we only collect from a given router. Therefore, as a first step, we collect *per-packet raw inputs* provided by the router: for instance, for each packet, both at the ingress and at the egress, we can record the packet's (1) *flow 5-tuple* (source IP address & port, destination IP address & port, and protocol), (2) captured *arrival time*, (3) *packet length*, (4) TCP timestamp *TSVal* set by the flow source (when available), and (5) *packet sequence number* from the TCP header.

Per-slot inputs. Directly feeding the above per-packet raw inputs into the Dragonfly system raises several issues. In particular, a sudden large burst of packets from a monitored flow could temporarily require a classification processing rate that is above the Dragonfly capacity, and therefore jam the system and degrade its real-time properties. Instead, we choose to aggregate the per-packet raw inputs into *per-slot inputs*, e.g., to obtain one input per monitored flow every 1 ms. This design choice clearly loses some information in the aggregation process, to the benefit of an increased reliability in the processing speed, which increases Dragonfly's readiness for real-life implementation.

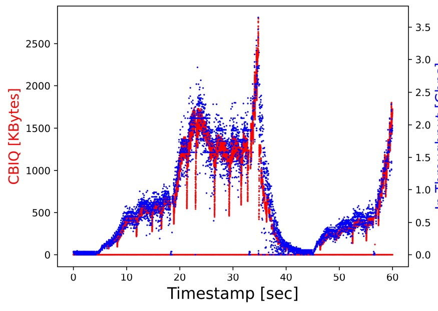
In the Dragonfly system, we implement the following approximate per-slot inputs for each monitored flow, based on the above per-packet raw inputs: (1) *ingress throughput*, i.e., number of packets or number of bytes arrived for this flow in this slot, (2) similarly, *egress throughput*, (3) estimated *number*



(a) Reno



(b) CUBIC



(c) BBR

Fig. 1. CBIQ vs. throughput: Experiments illustrating the *CBIQ* (in red, left axis) and *ingress throughput* (in blue, right axis) measures for a given flow of three CCA types: (a) Reno, (b) CUBIC and (c) BBR. All experiments run on a physical network with 15 additional background flows (5 Reno, 5 CUBIC and 5 BBR) and measures are collected every 1 ms. The CBIQ plots are relatively stable and present characteristic shapes, while the throughput measures tend to fluctuate between quantized values of n packets per ms with higher variability.

of packet drops in this slot because of a full buffer (this is only an estimate based on the sequence numbers, not a direct measure), (4) estimated *CBIQ (Connection Bytes in Queue)*, which we define as the number of bytes in the shared queue that belong to this flow at the end of the slot, (5) *maximal source time gap*: for each incoming packet, we compute the TSVal difference with the previous packet from the same flow (i.e., for the n^{th} packet of a flow, we compute TSVal(n)-TSVal($n-1$)), and later take the maximum among all packets in this period, and (6) *maximal arrival time gap*, computed in the same manner but based on the capture timestamp upon arrival at the router rather than on the source timestamp. Note that aggregating raw data is not novel. DeePCCI [31] relies on a measure similar to the per-slot ingress throughput, and Seiðr [32] relies on an input akin to the histogram of the capture arrival time gap.

B. CBIQ

Need for Built-in Memory. As detailed below, a key insight regarding the above design goals of the Dragonfly algorithm is that they favor the use of an input measure that can "remember" the past. In other words, the input measure at some time t should also provide us some indication on what happened before t. This is a key reason behind our use of CBIQ: the queue size devoted to a given flow reflects its cumulative arrival process (cf. Lemma 1.1 [41]), i.e. an integral of part of its past throughput. Therefore, even though we measure it at a single point of time, it implicitly tells us something on its past throughput.

We are interested in "remembering" the past for two reasons: First and foremost, since we intend to catch a flow *in-flight*, i.e., analyze it starting from an arbitrary point in time and provide real-time online classification, we cannot rely on using measures of its past. We can only measure from the point we focus on it. As a result, we do not have direct access to such measures as its past throughput. Similarly, any memory scheme that we could add to a neural network (e.g., LSTM) cannot tell us about what happened before it starts computations. On the other hand, by relying on an implicit memory-based measure, CBIQ provides us from the start with some limited information about the past, going back several RTTs. For instance, assume that we start measuring an arbitrary flow at an arbitrary time t, and that the CBIQ value at t is high. Then this immediately tells us that there is a good chance that the considered flow was more aggressive in the recent past, since it takes a large portion of the shared buffer. This immediately strengthens the probability that the flow uses for example BBR rather than CUBIC or Reno. As a result, CBIQ can provide us useful information even if we only measure it for a short time.

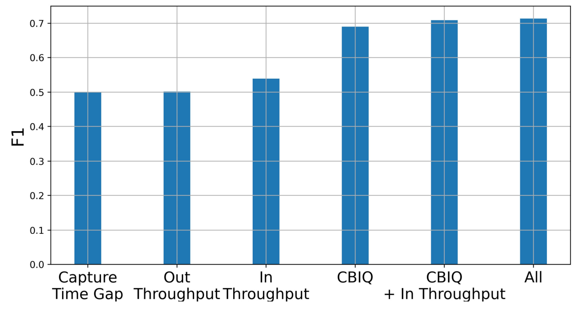
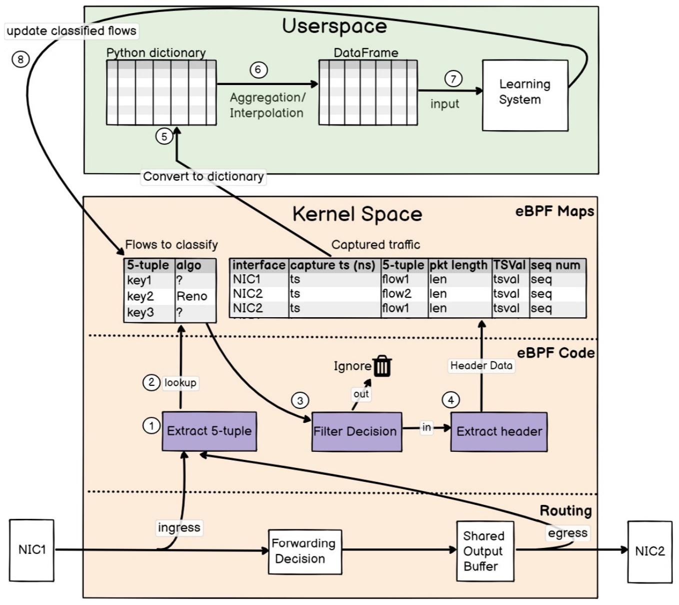


Fig. 2. Classification performance in a physical-network router, based on different inputs, using a random 100 ms sub-interval.

In addition, the built-in memory in CBIQ helps us because it can reduce the *volatility* of the input. Fig. 1 illustrates the evolution of CBIQ vs. that of the ingress throughput. We can see that CBIQ is less volatile (noisy). This helps us in two ways. First, we can see for instance how CBIQ exhibits a near-linear increase with Reno vs. a cubic shape with CUBIC. On the other hand, the large noise in the throughput measure makes it difficult to distinguish the two algorithms. Second, if we are to use random sampling of packets in order to increase the throughput of the system, then a less volatile sampled measure is expected to be more resilient to the sampling noise as the sampling rate increases.

Experiments. Fig. 2 illustrates the typical classification performance of the Dragonfly neural network when it is forced to rely on each of the above per-slot inputs. We first select the four inputs that perform best in Mininet simulations. We



5-tuple	algo
key1	?
key2	Reno
key3	?

interface	capture ts (ns)	5-tuple	pkt length	TSVal	seq num
NIC1	ts	flow1	len	tsval	seq
NIC2	ts	flow2	len	tsval	seq
NIC2	ts	flow1	len	tsval	seq

Fig. 3. eBPF-based Dragonfly architecture in a Linux router. Using a dynamic filter table of monitored flows that need to be classified, the eBPF code captures, filters, analyzes and saves the information of ingress and egress packets in kernel space. It then copies the raw per-packet input information to a userspace Python dictionary, before aggregating it into a dataframe with per-slot inputs. This dataframe serves as the input to the learning system.

then keep these four inputs and run experiments in a physical network. We train and test the classifier using hundreds of one-minute runs that implement a wide variety of settings. In each run, we send three special flows, implementing Reno, CUBIC and BBR. We vary the number of competing flows and the buffer size of the bottleneck-link router (as detailed in §IV-B). We then choose to classify each special flow based on a random sub-interval of 100 ms.

The experiments confirm our intuition. CBIQ is the best performing input, while ingress throughput and egress throughput are second and third, respectively. Furthermore, adding more inputs to CBIQ only yields a marginal benefit. Since we want a generic and reliable solution, we decide to employ the top two inputs in the Dragonfly system, i.e., CBIQ and ingress throughput. However, in the results below, we also compare against a Dragonfly solution that only employs CBIQ. Note that in all these experiments, all packets have the same size and TSO is disabled, so byte-based and packet-based throughput measures obtain identical performance.

TSO. As in the literature, we start by implementing the ingress throughput by using the number of incoming packets per time-slot [31]. However, we find that the performance of ingress throughput drops drastically when used in a software router that relies on TSO (TCP Segmentation Offload). This is because the packets are aggregated, and therefore the internal number of aggregated ingress packets does not reflect the real number of incoming packets. As a result, we turn to using *byte-based* counts of ingress throughput per time-slot. We find that it solves the TSO issue.

C. Scalable Traffic Collection

tcpdump challenges. We start by collecting the inputs using the common tcpdump [42]. However, we face two major challenges in light of our above design goals. First, tcpdump is too slow in processing traffic. We find that a 1-minute traffic at 1 Gbps takes 4 min just to extract the headers, even with a strong Core i7 10th-gen. processor. Second, we want the traffic collection filter to help the system analyze only selected flows, and ignore the remaining traffic. Third, we want to be able to quickly and dynamically change the list of flows to analyze. Unfortunately, tcpdump relies on the libpcap library [42], and traditional traffic analysis libraries such as libpcap and Scapy [43] use BPF (Berkeley packet filter) as a filtering method. BPF filtering is static, and therefore inappropriate.

eBPF. As a result, we select eBPF (extended BPF) as our filtering solution. eBPF can extract protocol header fields faster than libpcap, and can use lookup tables and C code to make a dynamic filtering decision.

Fig. 3 shows the eBPF architecture in a Linux router. We start by defining a slot time of 1 ms and a packet capture time of N ms (e.g., $N = 100$). Every N ms, we update a list of flows to monitor, and apply the following steps.

Phase 1. We start the packet capture for N ms. For every ingress and egress packet: (1) In kernel space, extract the flow's 5-tuple values from the IP and TCP headers. (2) Lookup in the first eBPF map whether the packet belongs to a monitored flow. (3) Decide accordingly whether the packet should be monitored. (4) If so, extract some more values from protocol headers, such as packet size and capture timestamp, and store them in a second, packet-capture eBPF map.

Phase 2. After N ms, conceptually stop the capture for this period, then: (5) Convert the format of the packet-capture eBPF map to a userspace raw-data Python dictionary. (6) Use the Python dictionary to create a Python DataFrame of per-slot inputs (cf §III-A and §III-D). (7) Use the DataFrame as input for the learning system, and classify the flows. (8) Update the list of classified flows in the first eBPF map. (9) Empty the packet capture eBPF map and start over.

D. Dragonfly Input Computation

Challenges. One of the challenges for the Dragonfly classification system is to compute the per-slot inputs based on the raw per-packet inputs. In an online real-time classification, unique challenges arise: (1) we wish to classify a flow based on an arbitrary period of time that does not necessarily contain the flow start period, and therefore we lack an initialization value, (2) we wish a fast $O(1)$ computation for the input values at each slot, independently of the flow rate, and (3) in case of random sampling, we may lack many of the packets.

Online computation without random sampling. Table I illustrates an online real-time computation of CBIQ and of the ingress throughput. We assume that it starts at time 101 (ms), i.e., 100 ms after the flow start. We want to provide a fast $O(1)$ computation at each slot.

CBIQ. At each slot t, we compute $CBIQ(t)$, the number of bytes of the monitored flow that have entered the shared buffer

TABLE I. Example of online Dragonfly input computation, with and without random sampling. (a) illustrates the per-packet raw inputs, with the sampled packets shown in bold. (b) illustrates the resulting CBIQ and ingress throughput computations, without sampling (left) and with sampling (right). With sampling, computations that rely on interpolation are underlined.

a Per-packet raw inputs

Ingress			Egress		
Time	Sequence	Length	Time	Sequence	Length
101	**1600**	100	102	**1300**	100
101	1700	200	102	1400	200
102	1900	100	103	1600	100
102	**2000**	100	103	**1700**	200
102	2100	100	104	1900	100
103	2200	100	106	2000	100
103	**2300**	100	106	**2100**	100
105	2400	100			
106	2500	100			
106	**2600**	100			

b Per-slot computed inputs

	Without Sampling				With Sampling			
	CBIQ			Thr.	CBIQ			Thr.
Time t	InSeq(t+1)	OutSeq(t+1)	CBIQ	In Thr.	InSeq(t+1)	OutSeq(t+1)	CBIQ	In Thr.
101	1900	1300	600	300	2000	1300	700	300
102	2200	1600	600	200	2300	1700	600	100
103	2400	1900	500	0	$\underline{2400}$	$\underline{1833}$	567	100
104	2400	2000	400	100	$\underline{2500}$	$\underline{1967}$	533	100
105	2500	2000	500	?	2600	2100	500	?

and not yet departed by the end of the slot. Unfortunately, since we do not have the raw input for the start of the flow, we do not even know the initial CBIQ at the start of the sub-interval, and cannot assume that $CBIQ(t) = 0$ at the start. Furthermore, we could somehow approximate CBIQ at the start of the sub-interval, then compute at each slot the total number of incoming and outgoing bytes to estimate the evolution of CBIQ. However, this can be an issue at high flow rates: the computation time and needed raw-input dictionary size increase with the number of packets in the slot (§IV-D).

Instead, we adopt an alternative approach. For each slot $t+1$, at the ingress (resp. egress), we record the sequence number of the first byte of the first arrived (resp. departed) packet, and denote it as $InSeq(t+1)$ (resp. $OutSeq(t+1)$). We then approximate the last byte arrived (resp. departed) by the end of the previous slot t using $InSeq(t+1)-1$ (resp. $OutSeq(t+1)$). Then, we approximate CBIQ as their difference: $CBIQ(t) \simeq \max([InSeq(t{+}1){-}1]-[OutSeq(t{+}1)-1],0) = \max(InSeq(t+1) - OutSeq(t+1), 0)$. Intuitively, the bytes in this difference are the bytes that have entered the queue and not departed, thus representing the queue size. We neglect losses and retransmissions, and assume in-order transmission, so this is only an approximation.

Table Ia illustrates an additional challenge: the next slot may not have an arrival (resp., departure). Therefore, we would need to use the first arrived (resp., departed) packet of the next *non-empty* slot. For instance, in the left-side, unsampled part of Table Ib, since there is no arrival in slot 104, we use $InSeq(103+1) = InSeq(104+1) = 2400$. However, spending time to look for this next non-empty slot would lose the $O(1)$ property. Instead, we proceed as follows, relying on our eBPF implementation of a sampling module that provides a raw-input dictionary that only keeps the first packet of each slot (§III-C). Such a raw-input dictionary of N slots would have at most N packets. For instance, when using 1-ms slots, a 100 ms dictionary would contain at most 100 packets. Then we can update $InSeq(t)$ and $OutSeq(t)$ going backwards from the last slot to the first, simply filling in empty values with the last seen value. This is an $O(1)$ task per slot.

Note that we do not seek to rely on drop information because of our design goal of minimal information (§II), and therefore we have to neglect the impact of losses.

Throughput. To compute the throughput at time t, we implement two approaches. The first is to simply add the bytes of all packets arrived at time t at the ingress. The second, whose per-slot complexity is $O(1)$ as for CBIQ, is to compute the ingress throughput $InThr(t)$ using $InThr(t) = \max(InSeq(t{+}1) - InSeq(t), 0)$. While faster, the second approach is more error-prone when there are many retransmissions.

Online computation with random sampling. The right side of Table Ib illustrates the online computation of CBIQ and of the ingress throughput when only the bold packets in Table Ia are sampled (using the random sampling presented in §III-C). We compute CBIQ and throughput based on $InSeq(t+1)$ and $OutSeq(t+1)$ as previously, but now need to take into account that we are missing many packets that introduce a larger error in these values.

Interpolation. To reduce the error, when there is no incoming (resp. departing) packet in slot $t+1$, we linearly interpolate the value of $InSeq(t+1)$ (resp. $OutSeq(t+1)$) from the non-empy slots that occurred last and will occur next. For instance, in Table Ib, $InSeq(102) = 2300$ and $InSeq(105) = 2600$, so we estimate that $InSeq(103) - 1 = 2400$ and $InSeq(104) = 2500$ (shown underlined). In practice, to keep an $O(1)$ computation, we start again from the final slot and go backwards. Each time we encounter a sequence of x empty slots, we go through it once to find the next non-empty slot and compute the interpolation slope, then a second time to fill the slot values, thus keeping an amortized $O(1)$ per-slot complexity over the raw-input dictionary.

E. Dragonfly Learning System

Overview. We are now ready to describe the Dragonfly learning system. After collecting per-packet traffic information using eBPF (§III-C), we provide the resulting dataframes to the learning system (Figure 3), and use these dataframes to compute the per-slot learning inputs (§III-A): CBIQ and ingress throughput (§III-B). To do so, we apply the algorithms described in §III-D. Once we collect these inputs, we feed them into the Dragonfly neural network described below, and finally obtain the classification result.

Inputs and outputs of the learning system. Fig. 4 provides an overview of the Dragonfly learning system. Assuming for instance that we use two per-slot inputs (CBIQ and ingress throughput) for each 1 ms slot, and that we want to classify

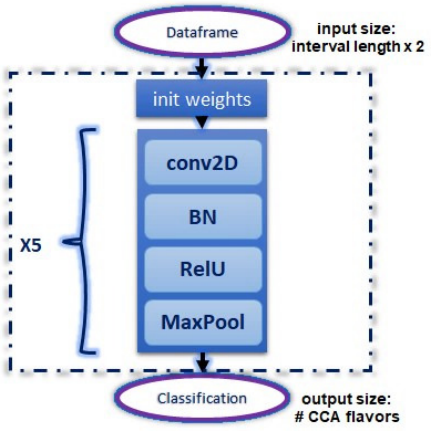


Fig. 4. Details of the CNN architectural layers.

a flow based on an arbitrary sub-interval of 100 ms, then the input dataframe size is 100 (slots per sub-interval) $\times$ 2 (input features). We normalize each dataframe by dividing each input value by its maximum value in the dataframe, thus obtaining a normalized value in $[0, 1]$. Then, at the end of the classification system, we obtain a number of outputs equal to the number of classified CCA flavors. For instance, if we have 3 CCA flavors, then we get 3 output values. The highest value corresponds to the most likely CCA flavor as determined by the system.

CNN overview. Using the pytorch library, we have implemented a CNN (convolutional neural network) module for automatic CCA classification given the above inputs and outputs. We choose the CNN architecture rather than (1) a more complex RNN (recurrent neural network) with an LSTM (long-short term memory) or GRU (gated recurrent unit) [44], since we want a simple approach that is easily implementable in routers; and (2) a basic fully-connected neural network, in order to exploit the *temporal locality* in the dataframes, i.e., the correlation between close time-slots, which are expected to be relatively similar, while keeping a lightweight architecture.

As illustrated in Fig. 4, to build the CNN, we choose a kernel of size 3×2, 50 convolutional filters, and 5 subsequent modules, each consisting of the following: a 2D convolution layer, a 2D batch normalization (BN), a ReLU activation, and a 2D MaxPool layer. The CNN outputs an array of numbers that indicate the classification score of each potential CCA. The maximum number corresponds to the predicted CCA.

CNN convolutional layers. The initial convolutional layer is designed to capture patterns in the input data. The kernel size is set as (3,2), where 2 corresponds to the number of input features (CBIQ and throughput), thus aligning with the structure of the input data. The convolutional operation computes the dot product between the input data and learnable filters, capturing relevant features specific to the target scenario. Batch normalization ensures stable training, and the ReLU activation introduces non-linearity to the model.

CNN max-pooling layers. Max-pooling layers reduce the data spatial dimensions, focusing on the most salient features. The kernel size and stride grow with the interval size.

CNN final convolutional and linear layers. The final convolutional layer produces output channels needed for the classification. The subsequent operations, including mean and squeeze, process the output to obtain a number of outputs that is equal to the classified number of CCA flavors.

Training. We train the model once offline, then reuse it online without retraining it. To train the Dragonfly CNN, we first run 1,500 experiments, either for 10 seconds (Mininet emulation) or for a minute (physical network). These experiments have variable settings to make the training more robust, as detailed in §IV. Assume that we intend to learn to classify flows based on a 100-ms sub-interval. Then we pick a random sub-interval of 100 ms from each of these experiments, thus collecting 1,500 samples, and build their corresponding 1,500 input dataframes. We group those dataframes into batches of size 32 (thus obtaining some 47 batches). We then use some random 70% of the total input for training and the rest (30%) to testing. Training lasts up to 100 epochs, where at each epoch we check once each of the assigned batches, and can stop earlier in case of early convergence of the accuracy measure. We rely on typical optimization tools: an Adam optimization algorithm for stochastic gradient descent, a cross-entropy stopping criterion, and a StepLR learning rate schedule.

Training speed. In order to train the model, it takes roughly a second per epoch (during the backpropagation technique), assuming a 100-ms sub-interval and a 1-ms sampling time. Thus, training 100 epochs lasts some 100 seconds.

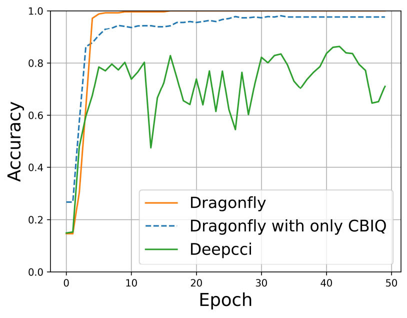
IV. Experiments

In this section, we run four types of evaluations. First, we start by implementing Dragonfly in a Mininet network emulator (§IV-A). Second, we implement Dragonfly on a physical-network testbed (§IV-B). We show that it presents similar results to Mininet, and study the impact of the classification time, amount of noise and sampling rate on classification performance. Third, we evaluate the performance of Dragonfly when communicating with remote destinations in the cloud, using high RTTs and conflicting with unknown flows (§IV-C). Finally, we evaluate the speed of the Dragonfly eBPF traffic collection using an incoming rate of 20 Gbps (§IV-D).

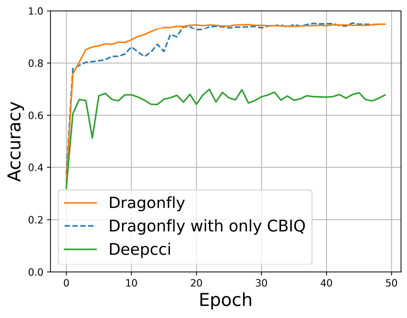
A. Mininet

CCAs. For a fair comparison, we follow the DeePCCI paper [31] in choosing the CCAs we try to classify: Reno, CUBIC and BBR. We consider $3(n+1) = 3n+3$ source nodes: three special sources, each generating either a Reno, CUBIC or BBR flow, together with $3n$ background flows comprised of n Reno flows, n CUBIC and n BBR. The background flows compete for the shared buffer utilization in the first bottleneck link and therefore can make the classification task harder. We vary the $3n$ within the set $\{0, 15, 30, 60, 75\}$. We use the iperf3 traffic generator to generate all the flows.

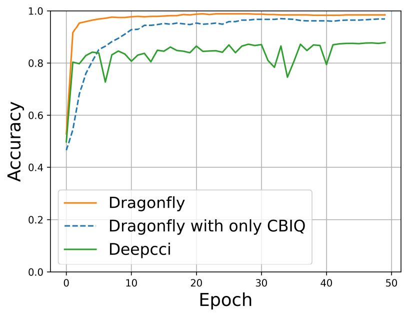
Run time. Each run lasts $x + 60$ seconds, where x is some initialization length that is drawn u.a.r. (uniformly at random) between 0 and 10 seconds. The starting time of each flow is then drawn u.a.r. between 0 and x.



(a) No background flows



(b) UDP background flows



(c) TCP Vegas background flows

Fig. 5. Training in Mininet: Accuracy as a function of training time, given three sources using Reno, CUBIC and BBR, with either (a) no background traffic, (b) UDP background, or (c) TCP Vegas background. In all cases, Dragonfly reaches the highest accuracy. Dragonfly restricted to CBIQ as a single input, without the additional ingress-throughput input, is close behind, indicating the large contribution of the CBIQ input. In contrast, DeePCCI presents a lower accuracy and takes a longer time to converge.

Topology. We deploy a network topology that connects the $3n+3$ sources to a shared destination through two successive routers. The first router is a bottleneck, and we use its local information to classify flows. The second router is a non-bottleneck pass-through router with empty queues. Both for training and testing, we run each evaluation hundreds of times by varying the topology settings (§III-E). The $3n+3$ sources are connected to the first bottleneck link using links of a variable rate between 100 Mbps and 1 Gbps, with propagation time in both ways. The two routers are connected using a 100-Mbps link with a 10-ms propagation time, thus creating a bottleneck at the first router. The second router connects to the shared destination using a link with a variable rate between 100 Mbps and 1 Gbps with 25-ms propagation time. The bottleneck link's shared buffer size is 500 packets.

Metrics. As usually done in the literature, we measure classification performance by using F_1, the harmonic mean of precision and recall. It is between 0 and 1, where 1 is best.

Algorithms. We implement (1) Dragonfly, (2) Dragonfly with only CBIQ as input and not the ingress throughput, to determine the part that CBIQ plays in the classification, and (3) DeePCCI [31] as a comparison baseline (DeePCCI has no open implementation). Except for the random-sampling experiment, we always run first-packet sampling for Dragonfly but no sampling for DeePCCI, even if it puts Dragonfly at a slight disadvantage. Unless mentioned, each algorithm is trained for each interval duration using the testing environment.

Dragonfly training time. Fig. 5 illustrates the impact of the offline training time on the accuracy of the Dragonfly system in Mininet. We remind that we train once offline then keep reusing the same model online without retraining. First, Fig. 5(a) shows how we train our three algorithms assuming that we only have one Reno flow, one CUBIC flow and one BBR flow sharing the buffer, and no background flows. We can see how the full Dragonfly system converges to an accuracy of 0.99, while Dragonfly that only uses CBIQ as input (without the ingress throughput) converges to 0.95. On the other hand, DeePCCI does not manage to converge and oscillates between 0.6 and 0.85. Moreover, we can see that the full Dragonfly system is the slowest to converge at the start, which is probably due to the fact that it relies on more parameters. Fig. 5(b) and Fig. 5(c) illustrate the impact of training time on accuracy when we add either UDP or TCP Vegas background flows, respectively. In the case of UDP, a background source host sends UDP packets of 1.4 KB at an average rate of 1,000 UDP packets per second. In the case of TCP Vegas, there are 10 TCP Vegas background hosts. In both cases, the results appear similar. The accuracy of Dragonfly is 0.95 with UDP background traffic, and 0.98 with TCP Vegas background. The volatility of DeePCCI decreases, but its accuracy is still lower than that of Dragonfly.

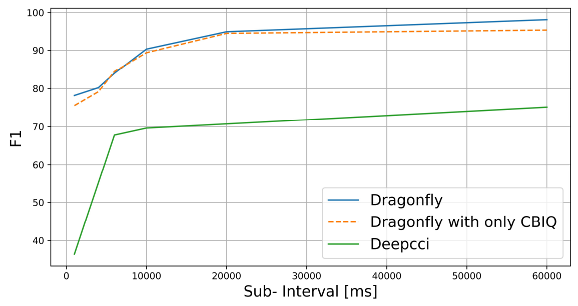


Fig. 6. Mininet: F_1 score performance as a function of the duration of the random flow sub-interval.

Impact of sub-interval duration. Fig. 6 illustrates the F_1 testing score as a function of the duration of the sub-interval randomly chosen to classify the flow. We examine a random interval of a varying size out of each 1-minute run, using hundreds of runs (§III-E). Dragonfly achieves an F_1 score of 0.97 for 1-minute intervals, while DeePCCI only reaches 0.75.

B. *Physical-Network Experiments*

Methodology. We now consider a physical-network testbed. We keep the methodology adopted within the Mininet emulation, with the following changes. First, unlike in Mininet, the link rates and propagation times cannot be arbitrarily changed, and we cannot arbitrarily add dozens of hosts. As a result, the $3n+3$ sources are implemented using two different

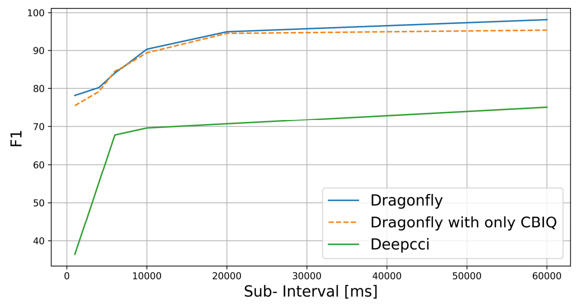


Fig. 7. Physical-network experiment: F_1 score performance as a function of the duration of the random flow sub-interval.

hosts with two network cards each. All the link rates in the physical network are 1 Gbps. All propagation times follow the physical layout of the setup and are under 1 ms. To change the training (and later testing) topology, we vary (1) the number $3n$ of background flows within the set $\{0, 15, 30, 60, 75\}$, as previously described, and (2) the buffer size of the Linux-based bottleneck link between 2K packets and 16K packets, causing a significant queueing delay of 25 ms to 200 ms. For each sub-interval duration, we train each algorithm once and use the same policy in all tests below without retraining.

Impact of sub-interval duration. Fig. 7 illustrates the F_1 score performance as a function of the sub-interval duration. We test a random interval of a varying duration out of each 1-minute run, using hundreds of runs. DeePCCI takes significantly longer to converge to a reasonable performance than Dragonfly. For instance, to reach an F_1 score of 0.7, it needs 18 seconds rather than 10 ms.

Impact of background traffic. Fig. 8 illustrates the F_1 score performance as a function of the number of background flows that compete with the three special flows. We use 10-second flow sub-intervals. As expected, the classification performance gradually decreases with the number of background flows. For instance, when going from 0 to 75 background flows, the F_1 score decreases from 0.9 to 0.74 for Dragonfly and from 0.77 to 0.56 for DeePCCI.

Impact of random sampling. Fig. 9 shows the F_1 score as a function of the percentage of packets dropped using random sampling. We use 10-second flow sub-intervals and no background flows. We consider random sampling as it is harder for Dragonfly: for instance, a first-packet-per-slot sampling that keeps the first packet of each slot and drops the others would not affect the performance of Dragonfly (§III-C). As expected, the classification performance gradually decreases as the sampling module drops more and more packets. Still, Dragonfly is relatively resilient to dropping about half the packets, and its F_1 score only decreases by about 0.1 even when randomly dropping 99% of the packets.

Sensitivity to CCA. Fig. 10 illustrates the F_1 score performance as a function of the flow sub-interval time that serves for classification, for each of the three CCAs of interest: Reno, CUBIC and BBR. In this experiment, each run lasts 10 seconds and we pick a random interval for the classification. The F_1 score increases for all CCAs a function of the interval length.

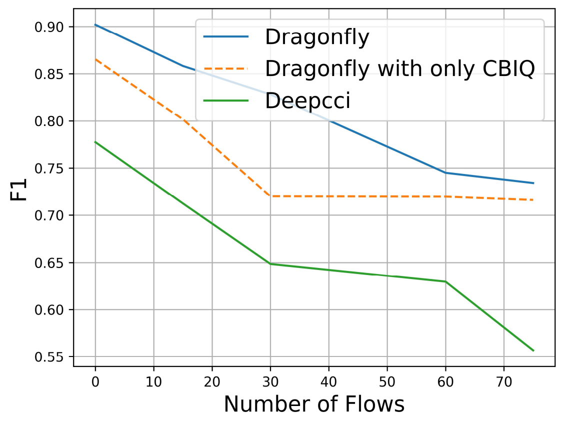


Fig. 8. Physical-network experiment: F_1 score as a function of the number of background flows. Performance gradually decreases with the number of competing flows.

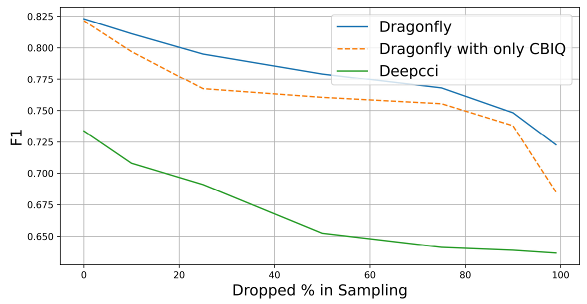


Fig. 9. Physical-network experiment: F_1 score as a function of the percentage of packets dropped using random sampling. As expected, the classification performance gradually decreases as the sampling module drops more and more packets.

BBR and CUBIC are classified more accurately than Reno, potentially because they have more packets that contribute to the classification.

C. Cloud Experiments

We now start from the above physical-network testbed but upgrade the line rates and select flow destinations in the cloud. Since the destinations are remote, propagation times are longer. Furthermore, there are background flows over which we have no control, competing in routers over which we also have no control.

Methodology. The methodology is similar to that of the physical network, with the following changes. We first install three AWS servers in Tokyo, N. California and Frankfurt. We measure a throughput of about 750 Mbps with an RTT of 300 ms to two of them, and a throughput of 850 Mbps with an RTT of 40 ms to a third. To do so, we dramatically increase the allowed TCP sending and receiving windows in all the source and destination machines. We use three servers as three flow sources. Each of the three servers independently picks any of the three CCAs in {Reno, CUBIC, BBR}, yielding 3^3 combinations, and chooses a distinct AWS server destination for its flow. Each server uses a distinct 10-Gbps line to connect

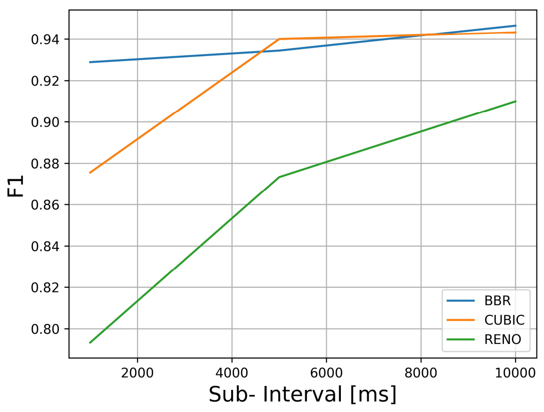


Fig. 10. Physical-network experiment: F_1 score for each CCA as a function of the classification sub-interval duration. BBR and CUBIC are classified more accurately than Reno, potentially because they are more aggressive and therefore have more packets that contribute to the classification.

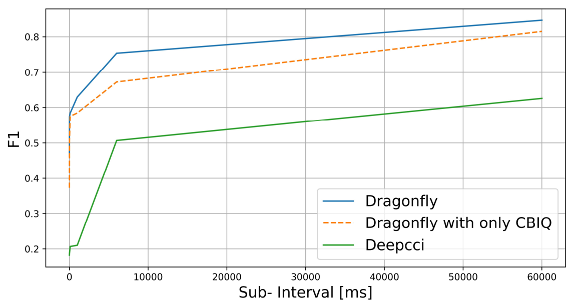


Fig. 11. Cloud experiment: F_1 score as a function of the duration of the random flow sub-interval, for flows sent to a cloud destination.

to a shared router, which uses a 1-Gbps line to get to the Internet. In addition, we vary the local router buffer length from 1,000 to 6,000 packets, yielding 6 settings, and run Dragonfly at the local router in real time. We run a total of $3^3 \cdot 6 = 162$ experiments, reflecting the 162 possible settings. We keep the same Dragonfly as trained on the physical network without retraining it.

Impact of sub-interval duration. Figure 11 plots the F_1 score performance as a function of the random-interval duration. As expected, Dragonfly outperforms DeePCCI. The outperformance is larger for very short intervals, where the built-in memory of CBIQ plays a large role. Specifically, the F_1 score of Dragonfly is 0.58 for 100 ms, 0.75 for 6 s, and 0.85 for a minute, while DeePCCI achieves 0.2, 0.5 and 0.62 accordingly.

D. Speed Experiments

Sampling. We find that in practice, the bottleneck to scaling the Dragonfly system to higher line rates is not in the classification speed, but in the quantity of traffic handled by the traffic collection, even after replacing tcpdump by eBPF (§III-C). To further scale the Dragonfly system and be able to classify more in-flight flows at extremely high rates, we add two sampling modules:

TABLE II. Speed experiment: one-minute experiment to check the eBPF traffic-collection rate with either 10-Gbps or 20-Gbps traffic, using an eBPF captured-traffic table size of 5M entries. We find that if eBPF does not apply first-packet-in-slot sampling at each 1-ms slot, the table fills up quickly and it cannot capture more traffic. Moreover, it takes 140 sec. to convert the table to a userspace dataframe. However, using first-packet sampling, eBPF can capture all traffic, and even build the dataframe $10\times$ faster than capture speed.

Settings			No first-packet sampling		First-packet sampling	
Rate	Flows	Packets	Capture duration	Build time	Capture duration	Build time
10 Gbps	2	24 M	13 sec (54%)	140 sec (0.09×)	60 sec (100%)	2 sec (30×)
20 Gbps	4	42 M	7 sec (12%)	140 sec (0.05×)	60 sec (100%)	6 sec (10×)

First-packet sampling. We implement in eBPF the option to only keep the first packet for each time slot, and drop the others. This reduces both the raw-input dictionary size and the computation complexity (as detailed in §III-D). To do so, for each incoming packet, we get the current time, round it to time-slot resolution, and look up the map. If there is a value for this flow in this slot, ignore the packet, else process it.

Random sampling. Sampling the first packet of each slot makes it easier for Dragonfly, because it does not alter the CBIQ estimation. To make things harder, we add a random-sampling module before our eBPF implementation. This module simply keeps a given percentage of the packets by selecting them uniformly at random, and drops the other packets of the flow. The traffic-collection code then needs to proceed using these packets only. Our goal is to check whether by sampling for example 10% of the packets, we are able to achieve a reasonable decrease in classification performance, in exchange for the $10\times$ decrease in the packet rate.

Speed experiment. We now want to evaluate whether our proposed eBPF first-packet-per-slot sampling module helps alleviate the traffic collection bottleneck. Table II details our experimental settings and results. We upgrade all of our physical-network testbed links to 10 Gbps. We assume four server sources, each sending one flow to one of two possible destination nodes through a unique shared router. Since all lines run at 10 Gbps and each router egress line is shared by two flows, each flow runs at about 5 Gbps. (In the 10-Gbps experiment, only two source servers are used with a shared destination.) A single eBPF router module collects all ingress and egress traffic at both router outputs. We run each experiment for a minute, with an eBPF captured-traffic table size limited to 5M entries, due to server memory limitations. We find that if eBPF simply collects traffic and does not apply first-packet sampling at each 1-ms slot, the table fills up quickly and we cannot capture more traffic. For instance, at 20 Gbps, it fills up after we collect 5M packets out of 42M, i.e., after $5/42 \cdot 60\,\text{s} \approx 7\,\text{s}$. Furthermore, it takes 140 s to later convert it to a userspace dataframe. However, using eBPF first-packet sampling, we can capture all traffic, and then build the dataframe $10\times$ faster, even though we rely on an unoptimized Python code. Note that this build time implicitly limits the number of monitored flows. To scale, we may want to use an optimized C implementation rather than a naive Python one. Moreover, note that the number of packets does not scale linearly with the line rate in the table, because TSO creates bigger packets more often at 20 Gbps.

V. Limitations

Sequence numbers. Throughout the paper, we consider TCP traffic with available sequence numbers, which help us accelerate the computations of the CBIQ and throughput inputs. However, these sequence numbers may not be available (e.g., with UDP or QUIC). Dragonfly still works with such traffic, but its speed decreases as the proportion of such packets increases, because in flows without available sequence number, we need to consider each packet to compute the per-slot input values. It is also possible to quickly update the per-slot input values upon each packet arrival, either using eBPF, a hash table or a programmable dataplane, requiring additional code.

Learned CCAs. Learned CCAs [14], [15], [45]–[48] raise new challenges because of their adaptive behavior. It is unclear whether a single generic classification framework can recognize them without adapting it to the details of each CCA.

New CCAs. Throughout the paper, we assume that Dragonfly has an existing list of CCAs to look for. On the one hand, Dragonfly does not need a specific detailed protocol specification of each new CCA, unlike some previous algorithms. It simply learns the behavior of this new CCA. It is designed to work on any new CCA that it can train for. On the other hand, given a new CCA that it has *not* trained for, there is no guarantee that Dragonfly will provide good results. We can only hope that it will try to assign it to the closest CCA in its database. We believe that it could classify together close families of CCAs with shared generic CCA properties, e.g. those that back off upon a loss or those that react to delay. However, this is beyond the scope of this paper.

Application improvement. In this paper, we focus on classifying CCAs. As mentioned in the Introduction, the result of this CCA classification can be used to improve performance in several ways, e.g., by fine-tuning the congestion signal to each CCA, buffering flows with different CCAs in different queues, or even re-routing hurtful flows. The performance improvement highly depends on the application, and goes beyond the scope of this paper.

VI. Related Work

Host classification. Several works [49], [50] attempt to identify the CCA used by a given end-host, such as a web server, often by actively influencing its traffic and monitoring the host's reaction. In this paper, we are interested in a passive method that may lie in a far-removed network router.

Offline classification. Several works [23]–[30] assume that they can monitor offline all the packets on both the forward packet path and the reverse ACK path. We do not assume access to the reverse path, due to potential routing asymmetry, and do not assume a full offline access to all of the flow's packets. On the other hand, our methodology does not enable us to provide fine-grained estimations of the CCA parameters.

DeePCCI [31] does not need to monitor the reverse path. It relies on a novel and fully generic classification approach that can handle any CCA. DeePCCI appears to be the state of the art in the literature, and we compare against it.

Online classification. In [32], the authors introduce an original way of leveraging programmable dataplanes. To our knowledge, it is the only paper in the literature that focuses on practical online classification at high line rates. However, it assumes additional hardware, and the accuracy of its CNN-based algorithm drops significantly when it needs to handle more than two CCAs, as would be expected in practice.

Enforcing fairness. In [38], the authors use programmable switches to classify flows into CCA families, e.g., modeling whether they are loss-based, delay-based, or both. Then, they group flows in buffers accordingly. They show a significant gain in fairness. However, they do not try to provide an exact CCA classification, and can classify in the same family two loss-based CCAs with significantly different aggressiveness.

Reverse-engineering. Reverse-engineering CCAs [51] could be seen as a more ambitious generalization of the classification task using program synthesis.

CCA properties. Using formal verification to establish CCA properties [52] can enable us to better understand CCAs. It could be seen as answering the dual question of what behavior can be obtained from a given CCA, whereas the classification task asks what CCAs could have caused a given behavior.

VII. Conclusion

In this paper, we introduced the Dragonfly system, which is designed to classify on the fly the congestion control algorithm of any flow that crosses a given router, starting at any time, and quickly provide a classification result with a reasonable accuracy. To do so, we introduced CBIQ, and explained how to compute it in $O(1)$. We further designed an eBPF-based implementation. Finally, we evaluated our Dragonfly system using a variety of platforms, showed how its accuracy clearly outperformed that of DeePCCI, and analyzed its traffic-collection speed using 20-Gbps traffic.

Acknowledgment

The authors would like to thank Roy Mitrany for his crucial help with the implementation. 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] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the network in cloud computing," in *ACM SIGCOMM*, 2012, pp. 187–198.

[2] R. Ware, M. K. Mukerjee, S. Seshan, and J. Sherry, "Modeling BBR's interactions with loss-based congestion control," in *ACM IMC*, 2019.

[3] D. Scholz, B. Jaeger, L. Schwaighofer, D. Raumer, F. Geyer, and G. Carle, "Towards a deeper understanding of TCP BBR congestion control," in *IFIP Networking*, 2018, pp. 1–9.

[4] J. Widmer, R. Denda, and M. Mauve, "A survey on TCP-friendly congestion control," *IEEE Network*, vol. 15, no. 3, pp. 28–37, 2001.

[5] S. Utsumi and G. Hasegawa, "Improving inter-protocol fairness based on estimated behavior of competing flows," in *IFIP Networking*, 2022.

[6] L. Brown, G. Ananthanarayanan, E. Katz-Bassett, A. Krishnamurthy, S. Ratnasamy, M. Schapira, and S. Shenker, "On the future of congestion control for the public internet," in *ACM HotNets*, 2020, pp. 30–37.

[7] A. Philip, R. Ware, R. Athapathu, J. Sherry, and V. Sekar, "Revisiting TCP congestion control throughput models & fairness properties at scale," in *ACM IMC*, 2021, pp. 96–103.

[8] P. Goyal, A. Narayan, F. Cangialosi, S. Narayana, M. Alizadeh, and H. Balakrishnan, "Elasticity detection: A building block for internet congestion control," in *ACM SIGCOMM*, 2022, p. 158–176.

[9] V. Arun, M. Alizadeh, and H. Balakrishnan, "Starvation in end-to-end congestion control," in *ACM SIGCOMM*, 2022, p. 177–192.

[10] K. Toledo, D. Breitgand, D. Lorenz, and I. Keslassy, "CloudPilot: Flow acceleration in the cloud," *Computer Networks*, 2023.

[11] C. X. Cai, F. Le, X. Sun, G. G. Xie, H. Jamjoom, and R. H. Campbell, "CRONets: Cloud-routed overlay networks," in *IEEE ICDCS*, 2016.

[12] A. Markuze, A. Bergman, C. Dar, I. Keslassy, and I. Cidon, "Kernels of splitting TCP in the clouds," *Netdev 0×14*, 2020.

[13] G. Zeng, J. Qiu, Y. Yuan, H. Liu, and K. Chen, "FlashPass: Proactive congestion control for shallow-buffered WAN," in *IEEE ICNP*, 2021.

[14] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira, "PCC vivace: Online-learning congestion control," in *Usenix NSDI*, 2018, pp. 343–356.

[15] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: A pragmatic learning-based congestion control for the internet," in *ACM SIGCOMM*, 2020, pp. 632–647.

[16] Y.-J. Song, G.-H. Kim, I. Mahmud, W.-K. Seo, and Y.-Z. Cho, "Understanding of BBRv2: Evaluation and comparison with BBRv1 congestion control algorithm," *IEEE Access*, vol. 9, pp. 37 131–37 145, 2021.

[17] L. Yu, J. Sonchack, and V. Liu, "Cebinae: scalable in-network fairness augmentation," in *ACM SIGCOMM*, 2022, pp. 219–232.

[18] A. Saeed, V. Gupta, P. Goyal, M. Sharif, R. Pan, M. Ammar, E. Zegura, K. Jang, M. Alizadeh, A. Kabbani *et al.*, "Annulus: A dual congestion control loop for datacenter and wan traffic aggregates," in *ACM SIGCOMM*, 2020, pp. 735–749.

[19] F. Yang, Q. Wu, Z. Li, Y. Liu, G. Pau, and G. Xie, "BBRv2+: Towards balancing aggressiveness and fairness with delay-based bandwidth probing," *Computer Networks*, 2022.

[20] H. Tian, X. Liao, C. Zeng, J. Zhang, and K. Chen, "Spine: an efficient DRL-based congestion control with ultra-low overhead," in *ACM CoNEXT*, 2022, pp. 261–275.

[21] Y. Ma, H. Tian, X. Liao, J. Zhang, W. Wang, K. Chen, and X. Jin, "Multi-objective congestion control," in *ACM EuroSys*, 2022.

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

[23] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, "On the characteristics and origins of internet flow rates," in *ACM SIGCOMM*, 2002.

[24] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, "Inferring TCP connection characteristics through passive measurements," in *IEEE Infocom*, vol. 3, 2004, pp. 1582–1592.

[25] J. Oshio, S. Ata, and I. Oka, "Identification of different TCP versions based on cluster analysis," in *IEEE ICCCN*, 2009, pp. 1–6.

[26] D. H. Hagos, P. E. Engelstad, and A. Yazidi, "Classification of delay-based TCP algorithms from passive traffic measurements," *IEEE NCA*, 2019.

[27] N. Ohzeki, R. Yamamoto, S. Ohzahata, and T. Kato, "Estimating tcp congestion control algorithms from passively collected packet traces using recurrent neural network." in *ICETE*, 2019, pp. 33–42.

[28] B. Nithya, V. Venkataraman, D. Nithin Balaaji, and C. Chud, "A CNN-LSTM approach for classification of major TCP congestion control algorithms," in *Intelligent Sustainable Systems*, 2022, pp. 579–591.

[29] T. Sawada, R. Yamamoto, S. Ohzahata, and T. Kato, "TCP congestion control algorithm estimation by deep recurrent neural network and its application to web servers on internet," *International Journal on Advances in Networks and Services*, 2023.

[30] E. Kfoury, J. Crichigno, and E. Bou-Harb, "P4cci: P4-based online tcp congestion control algorithm identification for traffic separation," in *IEEE ICC*, 2023, pp. 4007–4012.

[31] C. Sander, J. Rüth, O. Hohlfeld, and K. Wehrle, "DeePCCI: Deep learning-based passive congestion control identification," in *Workshop on Network Meets AI & ML*, 2019, pp. 37–43.

[32] K. A. Simpson, R. Cziva, and D. P. Pezaros, "Seiðr: Dataplane assisted flow classification using ML," in *IEEE Globecom*, 2020, pp. 1–6.

[33] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM CCR*, 1989.

[34] I. Stoica, S. Shenker, and H. Zhang, "Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks," in *ACM SIGCOMM*, 1998, pp. 118–130.

[35] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating fair queueing on reconfigurable switches," in *Usenix NSDI*, 2018.

[36] R. Mahajan, S. Floyd, and D. Wetherall, "Controlling high-bandwidth flows at the congested router," in *IEEE ICNP*, 2001, pp. 192–201.

[37] E. Zahavi, A. Shpiner, O. Rottenstreich, A. Kolodny, and I. Keslassy, "Links as a Service (LaaS): guaranteed tenant isolation in the shared cloud," in *IEEE JSAC*, 2019.

[38] B. Turkovic and F. Kuipers, "P4air: Increasing fairness among competing congestion control algorithms," in *IEEE ICNP*, 2020, pp. 1–12.

[39] M. Ibrar, L. Wang, N. Shah, O. Rottenstreich, G.-M. Muntean, and A. Akbar, "Reliability-aware flow distribution algorithm in sdn-enabled fog computing for smart cities," *IEEE Transactions on Vehicular Technology*, vol. 72, no. 1, pp. 573–588, 2022.

[40] (2023) Dragonfly code. [Online]. Available: https://github.com/DeanCarmel/Dragonfly

[41] A. J. Ganesh, *Big queues*. Springer, 2004.

[42] The Tcpdump Group. (2023) tcpdump & libpcap. [Online]. Available: https://www.tcpdump.org/

[43] P. Biondi and the Scapy community. (2023) Scapy documentation: performance of scapy. [Online]. Available: https://scapy.readthedocs.io/en/latest/usage.html#performance-of-scapy

[44] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[45] M. Schapira, "Network-model-based vs. network-model-free approaches to internet congestion control," in *IEEE HPSR*, 2018, pp. 1–8.

[46] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *ICML*, 2019, pp. 3050–3059.

[47] T. Meng, N. R. Schiff, P. B. Godfrey, and M. Schapira, "Pcc proteus: scavenger transport and beyond," in *ACM SIGCOMM*, 2020, pp. 615–631.

[48] W. Li, H. Zhang, S. Gao, C. Xue, X. Wang, and S. Lu, "Smartcc: A reinforcement learning approach for multipath TCP congestion control in heterogeneous networks," *IEEE JSAC*, vol. 37, no. 11, pp. 2621–2633, 2019.

[49] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu, "TCP congestion avoidance algorithm identification," *IEEE/ACM Transactions on Networking*, vol. 22, no. 4, pp. 1311–1324, 2013.

[50] A. Mishra, X. Sun, A. Jain, S. Pande, R. Joshi, and B. Leong, "The great internet TCP congestion control census," *ACM Measurement and Analysis of Computing Systems*, vol. 3, no. 3, pp. 1–24, 2019.

[51] M. Ferreira, A. Narayan, I. Lynce, R. Martins, and J. Sherry, "Counterfeiting congestion control algorithms," in *ACM HotNets*, 2021, pp. 132–139.

[52] V. Arun, M. T. Arashloo, A. Saeed, M. Alizadeh, and H. Balakrishnan, "Toward formally verifying congestion control behavior," in *ACM SIGCOMM*, 2021, pp. 1–16.

Dean Carmel received his M.Sc. in Electrical Engineering from the Technion, Israel, in 2022. His recent research interests include machine learning and congestion control.

Isaac Keslassy (M'02, SM'11) received his M.S. and Ph.D. degrees in Electrical Engineering from Stanford University, Stanford, CA, in 2000 and 2004, respectively. He is the Louis and Miriam Benjamin professor in Computer-Communication Networks at the Viterbi faculty of Electrical and Computer Engineering of the Technion, Israel. His research interests include the design and analysis of data-center networks and high-performance routers. He was the recipient of an ACM SIGCOMM test-of-time award and an ERC Starting Grant. He was associate editor for the IEEE/ACM Transactions on Networking.