Escala: Timely Elastic Scaling of Control Channels in Network Measurement

Hongyan Liu[1], Xiang Chen[123], Qun Huang[2], Dezhang Kong[1], Jinbo Sun[4],
Dong Zhang[3], Haifeng Zhou[1], Chunming Wu[1]

[1]Zhejiang University [2]Peking University [3]Fuzhou University
[4]Institute of Computing Technology, Chinese Academy of Science

IEEE INFOCOM 2022 - IEEE Conference on Computer Communications | 978-1-6654-5822-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOM48880.2022.9796830

***Abstract*—In network measurement, data plane switches measure traffic and report events (e.g., heavy hitters) to the control plane via control channels. The control plane makes decisions to process events. However, current network measurement suffers from two problems. First, when traffic bursts occur, massive events are reported in a short time so that the control channels may be *overloaded* due to limited bandwidth capacity. Second, only a few events are reported in normal cases, making control channels *underloaded* and wasting network resources. In this paper, we propose Escala to provide the *elastic scaling* of control channels at runtime. The key idea is to dynamically migrate event streams among control channels to regulate the loads of these channels. Escala offers two components, including an Escala monitor that detects scaling situations based on realtime network statistics, and an optimization framework that makes scaling decisions to eliminate overload and underload situations. We have implemented a prototype of Escala on Tofino-based switches. Extensive experiments show that Escala achieves timely elastic scaling while preserving high application-level accuracy.**

I. Introduction

Network measurement has been widely employed in modern networks. Its tasks are collaboratively executed by the data plane and control plane [1, 2, 3, 4]. In the data plane, these tasks are placed on switches to measure traffic at line rate. At runtime, they extract values (e.g., packet counts) from packets and generate *events*. Each event contains both the identifier (e.g., five-tuple) of a specific flow and the value associated with the flow. Then the tasks report events to the control plane via *control channels*, i.e., the links connecting the data plane and control plane. Here, we refer the stream of the events associated with a specific flow f as the *event stream* of f. In the control plane, several servers run network management applications to receive events and make corresponding decisions such as intrusion prevention [5].

However, today's network measurement suffers from two problems. (1) *Control channel overload*. The bandwidth capacity of a control channel is typically less than 100 Gbps [3, 4, 6]. Consider the case where a switch runs several sketches with different measurement purposes [5, 7]. Although each sketch employs a data structure with a limited size (e.g., a few MB [8, 9, 10, 11]), it usually defines a small time interval (e.g., 1 ms [12]) to report its data. In this case, the aggregated event rate of these tasks will exceed the bandwidth capacity of control channels. Thus, massive events will be lost, further affecting application-level accuracy. (2) *Control channel underload*. In most cases, the incoming traffic rate is only a few Gbps [13], such that only a few events are collected. Thus, the loads of several control channels may be far below their bandwidth capacity, decreasing energy efficiency.

To this end, we propose Escala, a framework that elastically scales control channels at runtime. Our key idea is to regulate the loads of control channels by migrating event streams between different channels. Specifically, when a control channel is overloaded, Escala selects a subset of event streams resided in the overloaded channel and migrates these streams to other channels, thus reducing the load of the overloaded channel. Moreover, when it detects an underloaded control channel, Escala migrates all the event streams resided in this channel to other channels. It then removes this channel to save resources.

However, we face two challenges to realize Escala. (1) We need to collect various types of network statistics to *detect scaling situations* (i.e., overload and underload situations). However, the variety of statistics makes it infeasible to collect various statistics in a uniform way, leading to significant user burdens. (2) We need to handle the diversity of statistics when incorporating these heterogeneous statistics to *make scaling decisions* (i.e., the decisions of migrating event streams).

To address these challenges, we propose two components in Escala. (1) An Escala monitor automatically collects necessary statistics from data plane switches and detects scaling situations based on these statistics. (2) An optimization framework makes scaling decisions to eliminate scaling situations. It encodes heterogeneous statistics and user-configurable thresholds as constraints. These constraints ensure that the output scaling decisions preserve the application-level requirements while minimizing the cost of migrating event streams.

We have implemented Escala on Tofino-based switches [14]. We conduct extensive experiments based on a real testbed and large-scale simulation to evaluate Escala. The results indicate that compared to other solutions, Escala achieves up to 68% reduction in the cost of migrating event streams and orders-of-magnitude reduction in the execution time.

II. Background and Motivation

This paper considers a typical network scenario compromising a data plane and a control plane. In the data plane, several switches process incoming flows and execute measurement tasks. In the control plane, network management applications such as heavy hitter detection handle network events. The two

Chunming Wu and Xiang Chen are corresponding authors.

planes communicate with each other via control channels. We consider the out-of-band measurement scenario where each switch connects to the control plane via a dedicated control channel.

A. Scaling Situations of Control Channels

Control channel overload. In extreme cases where traffic bursts arrive at switches, measurement tasks process packets at several Tbps [15] and generate massive events within a short time period. For example, in a Microsoft's data center, a 64-port switch, which port has a bandwidth capacity of 100 Gbps, runs a measurement task that selects one from every ten arrival packets as an event [6]. When the switch is fully saturated, its rate of processing packets reaches the peak, i.e., 64×100 Gbps. In this case, the task generates events at 64×100 Gbps $\times\frac{1}{10}$=640 Gbps. However, since each switch port is 100 Gbps, the bandwidth capacity of a control channel is also 100 Gbps. Thus, if sending events at 640 Gbps via a control channel, the channel will be unavoidably overloaded, leading to around 84% event loss. Such a high loss rate unavoidably decreases application-level accuracy, leading to wrong decisions (e.g., omitting malicious flows) [3]. A naive solution to handle control channel overload is to launch several channels. However, it requires several switch ports to connect to control channels, which inevitably reduces overall switch throughput.

Control channel underload. In normal cases, the incoming traffic rate is tens of Gbps [13]. Thus, several control channels and control plane servers are underloaded as they only need to handle a few events. It leaves room for improvement in resource efficiency. For instance, an idle server connected with an underloaded control channel consumes more than 50% of the power of a fully-loaded server in content delivery networks [16]. Thus, administrators can enhance resource efficiency by detecting and eliminating underload situations at runtime.

B. Related Works

However, existing solutions in network measurement fail to detect and eliminate scaling situations at runtime. We classify existing solutions into three types and discuss their limitations.

Event collection. There are several approaches for event collection. Sampling [17, 18, 6] only selects a few packets as events and thus reduces the load of collecting events. However, it unavoidably sacrifices accuracy. ApproSync [3] aggregates events in the switch ASIC to control the sending rate of events and avoid saturating control channels. *Flow [2] and Marple [19] buffer events in the switch ASIC before sending events to the control plane. RSS [20] only collects the events with higher information gain. However, these solutions are still limited by switch memory and control channel bandwidth. When traffic bursts arrive, massive events are generated and easily exhaust switch memory and the bandwidth of control channels, leading to unavoidable event drop. Also, these solutions fail to address the problem of control channel underload. TurboFlow [1] and Martini [12] offload applications onto the switch operating system (OS) so that they process events entirely inside the switch without involving control channels. However, such offloading suffers from the limitation of PCIe channels connecting the switch OS and the switch ASIC, which may still incur event loss. MTP [4] tries to eliminate overload situations via measurement task placement before runtime. However, MTP assumes that network topology is fixed. Thus, it fails to address scaling situations in dynamic networks [21].

Event migration. Some solutions aim to balance the loads of different switches. Swing State [22] migrates counter values within the data plane to achieve timely state synchronization among switches. P4Sync [23] secures the migration of Swing State with authenticity guarantees. P4State [24] only migrates essential values to reduce overhead. RoDiC [25] migrates some measurement operations from overloaded switches to spare ones. Some other solutions [26, 27, 28] migrate flow states among network functions. However, the above solutions fail to resolve the scaling situations of control channels.

Measurement systems. In recent years, many measurement systems have been proposed to serve network management. FCM-Sketch [29], UnivMon [9], and ActiveCM+ [30] support general measurement tasks with accurate and hardware-compatible sketches. These systems report both individual large flows and traffic distributions. Moreover, the method proposed by Romain Fontugne *et al.* [31] exploits traceroute data to measure link delay changes and network disruptions. FlowMon-DPDK [32] collects fine-grained statistics at both packet and flow levels. ANLS [33] dynamically tunes its sampling rate to enhance the accuracy of measuring small-size flows. Rhaban Hark *et al.* [34] proposed a measurement point selection scheme that maximizes the obtained network state information. However, the above systems cannot address scaling situations at runtime, leaving room for improvement.

III. Escala Overview

Goal. We aim to enhance network measurement with *elastic scaling* of control channels. Here, elastic scaling refers to the capability of migrating event streams among different control channels. Specifically, when detecting an overloaded control channel, elastic scaling is to *scale out* the overloaded channel by migrating some event streams resided in the channel to other channels. Moreover, when detecting an underloaded control channel, the elastic scaling is to *scale in* the channel by migrating all event streams resided in the channel to other channels and then removing the underloaded channel.

In particular, elastic scaling needs to achieve *low cost* in its migration. Specifically, applications typically assign a limited time budget, i.e., *deadline*, to the collection of event streams. For example, heavy hitter detection requires the collection to be completed within a few milliseconds [9, 12]. If the collection time exceeds the deadline, the application can only work on out-of-date events, thus failing to make timely and precise decisions in response to network anomalies. However, elastic scaling may bring additional latency cost due to its

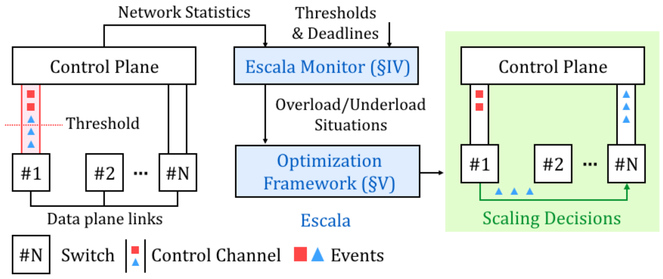


Fig. 1: Escala overview.

migration, leading to the risk of violating deadlines. Thus, our design should minimize the time of migrating event streams.

Escala. To achieve our goal, we propose Escala, a framework that achieves elastic scaling of control channels. As shown in Figure 1, we design two components, an Escala monitor (§IV) and an optimization framework (§V). The monitor offers automatic strategies to collect various statistics and detect scaling situations. For each situation, the framework leverages the collected statistics to make the cost-optimal scaling decisions that eliminate scaling situations. Specifically, it takes statistics, user-configurable thresholds, and deadlines as input. It formulates the objective and constraints of elastic scaling based on its input, and solves the scaling problem in polynomial time via an efficient heuristic.

IV. Escala Monitor

Escala monitor collects realtime statistics to detect scaling situations. If a scaling situation occurs, it delivers the collected statistics to the optimization framework for further processing.

Statistic collection. Escala monitor needs to collect a variety of statistics, including:

- *Statistics of control channels*. For the i-th channel c_i, there are three statistics, including the realtime load $l_c(c_i)$ (in Gbps), the transmission latency $t_c(c_i)$ (in nanoseconds), and the sending rate $r_{c_i}(s_j)$ (in Gbps) of the j-th event stream s_j resided in the i-th control channel c_i.
- *Statistics of data plane links*. For the k-th link e_k, there are two statistics, including the realtime load $l_l(e_k)$ (in Gbps), and the transmission latency $t_l(e_k)$ (in nanoseconds).
- *Statistics of switches*. For the switch u in the data plane, there are two statistics, including the realtime load $l_v(u)$ (in Gbps), and the transmission latency $t_v(u)$ (in nanoseconds).

However, the variety of statistics complicates statistic collection. In response, Escala monitor classifies the statistics into three types, including realtime loads, transmission latency, and the sending rates of event streams. For each type of statistics, it uses a dedicated strategy to collect them. We present its strategies as follows (assume there are Q switches, K links, N channels while the i-th channel transfers M_i streams).

① Realtime loads ($l_c(c_i)$, $l_l(e_k)$, and $l_v(u)$, $\forall i \in [1, N]$, $\forall k \in [1, K]$, $\forall u \in [1, Q]$). First, as Escala monitor works on the control plane, it has a full view of realtime status of control channels. Thus, the load $l_c(c_i)$ of the i-th channel c_i can be directly obtained by reading the status of c_i. Second, for the load $l_l(e_k)$ of the k-th link e_k, Escala monitor exploits per-port switch counters, each of which records the number of bytes transmitted via a specific switch port. It first locates the switch port at one end of e_k. Then it periodically queries the value of the counter attached to the port, and computes the divergence Δ of counter values between two queries. With Δ and the time interval t_{query} between two queries, Escala monitor calculates $l_l(e_k) = \frac{\Delta}{t_{query}}$. Third, for the load $l_v(u)$ of the switch u, Escala monitor accumulates the load of every ingress link connected to u as $l_v(u)$.

② Transmission latency ($t_c(c_i)$, $t_l(e_k)$, and $t_v(u)$, $\forall i \in [1, N]$, $\forall k \in [1, K]$, $\forall u \in [1, Q]$). First, for the latency $t_c(c_i)$ of the i-th channel c_i, Escala monitor modifies the heartbeat mechanism [35]. In this mechanism, the control plane periodically sends heartbeat packets to switches via control channels. If a switch works normally, it will immediately return heartbeat packets to the control plane. Thus, Escala monitor intercepts every heartbeat packet before sending these packets, and adds a new header field that records the current time t_{send} to the packet. When the packet is returned, Escala monitor extracts t_{send} from the packet and records the current time t_{rcv}. Thus, $t_c(c_i) = \frac{t_{rcv} - t_{send}}{2}$.

Second, for the latency $t_l(e_k)$ of the k-th link e_k, Escala monitor leverages link layer discovery protocol (LLDP) [36]. Specifically, it generates an LLDP packet and piggybacks the current time t_{send} on the type-length-value (TLV) fields of the packet. Then it sends the packet via the m-th control channel c_m. In the data plane, a switch u receives the LLDP packet and directly broadcasts the packet to every port. When its neighborhood switch v receives the packet, it will immediately deliver the LLDP packet back to the control plane. When the control plane receives an LLDP packet from the n-th channel c_n, Escala monitor extracts t_{send} from the packet and records the current time t_{rcv}. It computes $t_{rcv} - t_{send}$:

$$t_{rcv} - t_{send} = t_c(c_m) + t_l(e_k) + t_c(c_n)$$

where $t_c(c_m)$ and $t_c(c_n)$ denote the latency of the m-th channel and that of the n-th channel, respectively. Here, $t_c(c_m)$ and $t_c(c_n)$ can be obtained by the heartbeat mechanism. Thus, Escala monitor calculates $t_l(e_k)$ as:

$$t_l(e_k) = t_{rcv} - t_{send} - t_c(c_m) - t_c(c_n)$$

Note that the above result contains the processing latency of switches u and v. However, as switches only parse and broadcast LLDP packets without other operations, their latency of processing an LLDP packet is small ($< 1\mu s$ in general). Thus, the above equation returns accurate $t_l(e_k)$.

Escala monitor does not affect heartbeat and LLDP mechanisms. Also, it only consumes 6 bytes in packet headers to record t_{send}. Compared to a general maximum transmission unit (MTU) of 1500 bytes, the additional bandwidth overhead is 0.4%, which is relatively small.

Third, for the latency $t_v(u)$ of the switch u, Escala monitor generates a probe, which records t_{send}, and sends the probe to u via the channel c_i, which latency is $t_c(c_i)$. Also, it orders u to process the probe using the entire packet processing pipeline, and then route the probe to the control plane. When it receives the probe, it records t_{rcv} and calculates $t_v(u)$:

$$t_v(u) = t_{rcv} - t_{send} - 2 \cdot t_c(c_i)$$

③ Sending rates of event streams (i.e., $r_{c_i}(s_j), \forall i \in [1, N], \forall j \in [1, M_i]$). For each control channel, Escala monitor measures the sending rate of every event stream resided in the channel. Specifically, for the i-th control channel c_i, it classifies received events into different streams based on the flow identifiers recorded in these events. The events in an event stream record the same flow identifier. To measure the sending rate $r_{c_i}(s_j)$ of the j-th stream s_j resided in the i-th channel c_i, Escala monitor counts the number num of events received from s_j in a time interval t. It calculates $r_{c_i}(s_j)$ as $\frac{num \cdot size_{event}}{t}$, where $size_{event}$ refers to the size of an event.

Overload and underload situation detection. Escala monitor exposes two user-configurable thresholds, Θ_{top} and Θ_{bottom}, to administrators. The two thresholds specify the peak load of a control channel and the bottom load of a control channel, respectively. They can be flexibly configured with respect to dynamic network status. For example, we can set Θ_{top} and Θ_{bottom} to 80% and 20% of the bandwidth capacity of a control channel, respectively. Escala monitor uses the two thresholds to detect scaling situations. Specifically, it first calculates the current load $l_c(c_i)$ of the i-th channel c_i. $l_c(c_i)$ equals the accumulation of sending rates of all the event streams resided in c_i (assume that c_i has M_i event streams):

$$l_c(c_i) = \sum_{\forall j \in [1, M_i]} r_{c_i}(s_j)$$

Then it detects a scaling situation if one of the following inequalities holds (assume that there are N channels):

$$\text{Overload: } l_c(c_i) \geq \Theta_{top}, \exists i \in [1, N]$$
$$\text{Underload: } l_c(c_i) \leq \Theta_{bottom}, \exists i \in [1, N]$$

V. Optimization Framework

The optimization framework makes scaling decisions to eliminate scaling situations. When detecting an overloaded channel c_i, which load $l_c(c_i)$ exceeds the threshold Θ_{top}, the framework selects some event streams resided in c_i and migrates these streams to other channels until $l_c(c_i)$ is below Θ_{safe}. When detecting an underloaded channel c_i, which load $l_c(c_i)$ is below the threshold Θ_{bottom}, the framework migrates the event streams in c_i to other channels, and removes c_i to save resources. Table I presents the notation of symbols.

A. *Input: Network Statistics and User-Configurable Values*

The input of the optimization framework comprises both the statistics of substrate network, and user-configurable thresholds and deadlines of elastic scaling.

Network statistics. The network is represented by a directed graph $G = (V_G, E_G, C_G)$, where $V_G = \{u\}$ denotes the set of switches; $E_G = \{e_i\}$ denotes the set of data plane links connecting switches; $C_G = \{c_i\}$ denotes the set of control channels, each of which connects a switch with a control plane server. The network contains $Q = |V_G|$ switches, $K = |E_G|$ links, and $N = |C_G|$ control channels.

TABLE I: Notation of symbols

Symbol	**Description**
	Network statistics
Q, K, N	Number of switches, links, control channels.
ω_{c_i}	Set of event streams in control channel c_i.
M_i	Number of event streams in control channel c_i.
$r_{c_i}(s_k)$	Sending rate of event stream s_k in control channel c_i.
$b_c(c_i)$	Bandwidth capacity of control channel c_i.
$t_c(c_i)$	Transmission latency of control channel c_i.
$l_c(c_i)$	Realtime load of control channel c_i.
$b_v(u)$	Bandwidth capacity of switch u.
$t_v(u)$	Transmission latency of switch u.
$l_v(u)$	Realtime load of switch u.
$b_e(e_i)$	Bandwidth capacity of data plane link e_i.
$t_e(e_i)$	Transmission latency of data plane link e_i.
$l_e(e_i)$	Realtime load of data plane link e_i.
$P^{(u,v)}$	Set of paths between switch u and switch v.
$t_p(p)$	Transmission latency of network path p.
$b_p(p)$	Available bandwidth of network path p.
$\mathcal{E}(p, q)$	Variable indicating if link/switch q exists in path p.
$\mathcal{I}(c, v)$	Variable indicating if channel c is connected to switch v.
	User-configurable thresholds and deadlines
Θ_{top}	Overload threshold of control channel.
Θ_{bottom}	Underload threshold of control channel.
Θ_{safe}	Safe load threshold of control channel.
Φ_s	Deadline of event stream s.
	Variables of scaling decisions
$x^{c_i}(s_k)$	Variable indicating if stream s_k is selected to be migrated.
$y_{c_j}^{c_i}(s_k)$	Variable indicating if stream s_k is migrated from channel c_i to channel c_j.
$z_{c_j}^{c_i}(s_k, p)$	Variable indicating if path p migrates stream s_k from channel c_i to channel c_j.
$\alpha_{c_j}^{c_i}(s_k, p)$	Variable indicating the load of migrating s_k from channel c_i to channel c_j in path p.

- Each switch $u \in V_G$ has three statistics. (1) $b_v(u)$ represents the switch bandwidth capacity (in Gbps). (2) $l_v(u)$ represents the current load (in Gbps) of u. (3) $t_v(u)$ indicates the transmission latency (in nanoseconds) of u.
- Each data plane link $e_i \in E_G$ has three statistics. (1) $b_l(e_i)$ represents the bandwidth capacity (in Gbps) of e_i. (2) $l_l(e_i)$ denotes the current load (in Gbps) of e_i. (3) $t_l(e_i)$ denotes the transmission latency (in nanoseconds) of e_i.
- Each control channel $c_i \in C_G$ is associated with three statistics. (1) $b_c(c_i)$ represents the bandwidth capacity (in Gbps) of c_i. (2) $l_c(c_i)$ indicates the current load (in Gbps) of c_i. (3) $t_c(c_i)$ denotes the transmission latency (in nanoseconds) of c_i. We also use a boolean variable $\mathcal{I}(c, u)$ to indicate if the control channel c is connected to the switch u: $\mathcal{I}(c, u) = 1$ if c is connected to u; $\mathcal{I}(c, u) = 0$ otherwise.

The framework obtains the statistics of current loads and latency from Escala monitor. It acquires other statistics from the central controller that manages the entire network.

Moreover, it records a set $P^{(u,v)}$ of network paths between every pair (u, v) of switches ($\forall u, v \in V_G$). $P^{(u,v)}$ contains all the paths between u and v. A path is a sequence of switches connected by links. Here, we use a variable $\mathcal{E}(p, q)$ to indicate if a link or a switch $q \in E_G \cup V_G$ exists in a path $p \in P^{(u,v)}$: $\mathcal{E}(p, q)$=1 if q exists in p; $\mathcal{E}(p, q)$=0 otherwise. Each path $p \in P^{(u,v)}$ has two statistics. First, $t_p(p)$ denotes the transmission latency (in nanoseconds) of p, which is the sum of transmission latency of all the links and switches in p:

$$t_p(p) = \sum_{i=1}^{K}(\mathcal{E}(p, e_i) \cdot t_l(e_i)) + \sum_{\forall w \in V_G} (\mathcal{E}(p, w) \cdot t_v(w))$$

Second, $b_p(p)$ represents the available bandwidth (in Gbps) of the path p. It is the minimum among the available bandwidth of the links in the path:

$$b_p(p) = \min_{\forall i \in [1,K]} (\mathcal{E}(p, e_i) \cdot (b_l(e_i) - l_l(e_i))),$$
$$\forall u, v \in V_G, \ \forall p \in P^{(u,v)}$$

We elide the bandwidth capacity of switches because switches offer Tbps-level bandwidth [14], making switches almost impossible to become the bottleneck.

The framework also receives two types of statistics of event streams from Escala monitor. First, for the channel $c_i \in C_G$, we use a set $\omega_{c_i} = \{s_k\}$ to record the event streams resided in c_i. We use M_i to represent the size of ω_{c_i}, which equals the number of streams in c_i. Second, we use $r_{c_i}(s_k)$ to represent the sending rate of the stream s_k resided in the channel c_i.

User-configurable thresholds and deadlines. The framework accepts three thresholds for elastic scaling. Θ_{top} and Θ_{bottom} define the overload threshold and underload threshold of a control channel, respectively, as mentioned in §IV. Another threshold is Θ_{safe}. It indicates when to stop scaling out an overloaded channel. For example, assume that Θ_{top}=80 Gbps and Θ_{safe}=60 Gbps. For a 100 Gbps channel c_i, it is overloaded when its load reaches 80 Gbps. The strawman method is to directly halve the load of c_i by migrating at least 50% event streams in c_i to other channels. However, with the threshold Θ_{safe}, we could only migrate 25% streams to achieve more effective scaling. Moreover, the framework allows administrators to submit the deadlines of event streams. Each deadline corresponds to a stream associated with a specific flow (identified by five-tuple). For the event stream s_k, we use Φ_{s_k} to denote the deadline of s_k.

B. *Output: Scaling Decisions*

The output is the scaling decisions corresponding to a specific scaling situation. It contains four sets of variables.

- The set $\{x^{c_i}(s_k)\}$ indicates which event streams will be migrated. The variable $x^{c_i}(s_k)$ indicates whether the stream s_k in the channel c_i is selected to be migrated: $x^{c_i}(s_k) = 1$ if s_k is selected; $x^{c_i}(s_k) = 0$ otherwise.
- The set $\{y^{c_i}_{c_j}(s_k)\}$ specifies which channels to migrate selected streams to. The variable $y^{c_i}_{c_j}(s_k)$ indicates whether a selected stream s_k ($x^{c_i}(s_k) = 1$) in the channel c_i is migrated to another channel c_j: $y^{c_i}_{c_j}(s_k) = 1$ if s_k is migrated from c_i to c_j; $x^{c_i}_{c_j}(s_k) = 0$ otherwise.
- The set $\{z^{c_i}_{c_j}(s_k, p)\}$ indicates the paths that transfer selected streams to target channels. The variable $z^{c_i}_{c_j}(s_k, p)$ determines whether the path p transfers the selected stream s_k from the channel c_i to another channel c_j: $z^{c_i}_{c_j}(s_k, p) = 1$ if p transfers s_k; $z^{c_i}_{c_j}(s_k, p) = 0$ otherwise.
- The set $\{\alpha^{c_i}_{c_j}(s_k, p)\}$ specifies the assignment of the load of migrating event streams to network paths. The variable $\alpha^{c_i}_{c_j}(s_k, p)$ indicates that the load added to the path p when migrating the stream s_k from the channel c_i to the channel c_j. For example, assume the sending rate of s_k is 100 Gbps in c_i. When migrating s_k from c_i to c_j, we can assign 40% load of migrating s_k to a path p_a while assigning the remaining 60% load to another path p_b: $\{\alpha^{c_i}_{c_j}(s_k, p_a) = 40\,\text{Gbps}, \alpha^{c_i}_{c_j}(s_k, p_b) = 60\,\text{Gbps}\}$.

C. *Optimization Objective*

When making the scaling decisions, the framework aims to minimize the cost of migrating event streams. We choose the sum of migration time of all event streams as the cost because long migration time will delay the decision-making progress of applications. We formulate the objective as:

$$\min \sum_{\forall u,v \in V_G} \sum_{\forall p \in P^{(u,v)}} \sum_{\forall i,j \in [1,N]} (\mathcal{I}(c_i, u) \cdot \mathcal{I}(c_j, v))$$
$$\cdot \sum_{k=1}^{M_i} x^{c_i}(s_k) \cdot y^{c_i}_{c_j}(s_k) \cdot z^{c_i}_{c_j}(s_k, p) \cdot t_p(p) \qquad (1)$$

D. *Constraints*

① Effective migration. First, when the channel c_i is overloaded, its load $l_c(c_i)$ should be reduced below the safe threshold Θ_{safe} after migration:

$$l_c(c_i) - \sum_{i=1}^{N} \sum_{k=1}^{M_i} x^{c_i}(s_k) \cdot r_{c_i}(s_k) \le \Theta_{safe} \qquad (2)$$

Second, when the channel c_i is underloaded, its load $l_c(c_i)$ should be reduced to zero after migration:

$$l_c(c_i) - \sum_{i=1}^{N} \sum_{k=1}^{M_i} x^{c_i}(s_k) \cdot r_{c_i}(s_k) = 0 \qquad (3)$$

② Self-loop forbidden. The event streams in the channel c_i cannot be migrated to c_i:

$$y^{c_i}_{c_i}(s_k) = 0, \forall i \in [1, N], \forall k \in [1, M_i] \qquad (4)$$

③ Assignment of migration load. For the stream s_k in the channel c_i, the load of migrating s_k imposed on network paths should equal the sending rate $r_{c_i}(s_k)$ of s_k:

$$\sum_{\forall v \in V_G} \sum_{j=1}^{N} \mathcal{I}(c_j, v) \sum_{\forall p \in P^{(u,v)}} z^{c_i}_{c_j}(s_k, p) \cdot \alpha^{c_i}_{c_j}(s_k, p) = x^{c_i}(s_k) \cdot r_{c_i}(s_k),$$
$$\forall u \in V_G, \ \forall i \in [1, N], \ \forall k \in [1, M_i], \ \mathcal{I}(c_i, u) = 1 \qquad (5)$$

④ Load limitations of network paths. The migration of event streams must not overload network paths. Thus, for an arbitrary path p, the load assigned to p should be less than the available bandwidth of p:

$$\sum_{\forall i,j \in [1,N]} \mathcal{I}(c_i, u) \cdot \mathcal{I}(c_j, v) \sum_{k=1}^{M_i} z^{c_i}_{c_j}(s_k, p) \cdot \alpha^{c_i}_{c_j}(s_k, p) \le b_p(p),$$
$$\forall u, v \in V_G, \ \forall p \in P^{(u,v)} \qquad (6)$$

⑤ Load limitations of control channels. The migration of event streams must not cause new overloaded channels. Thus, for an arbitrary channel c_j, its load $l_c(c_j)$ should not exceed the overload threshold Θ_{top} after receiving migrated streams:

$$l_c(c_j) + \sum_{\forall u \in V_G} \sum_{i=1}^{N} \mathcal{I}(c_i, u) \sum_{\forall p \in P^{(u,v)}} \sum_{k=1}^{M_i} z_{c_j}^{c_i}(s_k, p) \cdot \alpha_{c_j}^{c_i}(s_k, p) < \Theta_{top},$$
$$\forall v \in V_G,\ \forall j \in [1, N],\ \mathcal{I}(c_j, v) = 1 \qquad (7)$$

⑥ Preserving deadlines. For the event stream s_k, its migration time plus the transmission latency of the channel to which s_k is migrated should be less than the deadline of s_k.

$$\max_{\forall p \in P^{(u,v)}} z_{c_j}^{c_i}(s_k, p) \cdot (t_p(p) + t_c(c_j)) < \Phi_{s_k}, \forall u, v \in V_G,$$
$$\forall i, j \in [1, N],\ \forall k \in [1, M_i],\ \mathcal{I}(c_i, u) \cdot \mathcal{I}(c_j, v) = 1 \qquad (8)$$

E. Problem Hardness

However, making scaling decisions is NP-hard.

Theorem 1. The problem of making scaling decisions with the objective in §V-C and the constraints in §V-D is NP-hard.

Proof. We prove Theorem 1 by considering a special case of the problem. The special case applies five restrictions that simplify our problem. First, it restricts that each network path has infinite bandwidth, i.e., $b_p(p) = +\infty$ ($\forall p$). Second, it restricts that the network paths for migrating event streams are determined, i.e., the set $\{z_{c_j}^{c_i}(s_k, p)\}$ is given. Third, it restricts that different network paths have the same transmission latency $t_p(p) = 1$ ($\forall p$). Fourth, it restricts that the deadlines of event streams are unlimited, i.e., $\Phi_{s_k} = +\infty$ ($\forall s_k$). Fifth, it restricts that the event streams selected to be migrated are determined, i.e., the set $\{x^{c_i}(s_k)\}$ is given. This special case adjusts the optimization objective to minimize the number of used control channels, which reduces our problem to an NP-hard bin-packing problem [37]. Thus, Theorem 1 follows. □

F. Heuristic Algorithm

The strawman method makes the optimal scaling decisions via commodity solvers. However, it typically spends several minutes due to problem complexity, which is unsuitable for online scaling. Thus, we design a heuristic, i.e., Algorithm 1, to obtain near-optimal scaling decisions in polynomial time.

Algorithm. Algorithm 1 first selects streams to be migrated. Then, for each selected stream s_k, it selects paths to migrate s_k and assigns the load of migrating s_k to these paths. Finally, it updates network statistics for the next selection.

① Select event streams (lines 1-12). Given an overloaded or underloaded channel c_i, Algorithm 1 records the event streams in c_i in a set ω_{c_i}. It sorts the streams in ω_{c_i} in the descending order of the deadlines attached to these streams (line 1). Next, it selects some streams in ω_{c_i} to migrate and records these streams in a stack Ω. When c_i is overloaded, it enumerates ω_{c_i}. For each stream $s_k \in \omega_{c_i}$, it pushes s_k to Ω until migrating the streams in Ω can decrease the load of c_i to the safe threshold Θ_{safe} (lines 3-8). When c_i is underloaded, it pushes all the streams in ω_{c_i} to Ω (lines 10-12).

② Select paths (lines 13-30). Algorithm 1 first locates the switch u connected to c_i. Then it records all the paths originated from u in a set P^u, and sorts P^u in ascending order of latency (lines 13-14). Next, it enumerates the streams in Ω. It extracts the stream s_k from the top of Ω (line 16). Then it extracts the first available path p from P^u (line 17). If p does not exist or migrating s_k via p will violate the deadline Φ_{s_k}, s_k cannot be migrated. This is because the network lacks of sufficient bandwidth resources to perform elastic scaling without violating user-specified deadlines. In this case, Escala reports this anomaly to administrators and terminates its execution (lines 18-19). Otherwise, p connecting u and another switch v is available. Thus, Escala locates the channel c_j connected to v (line 20). It marks that s_k is migrated from c_i to c_j via p (line 21). Then, it calculates the available bandwidth $\gamma(p)$ that can be used to migrate s_k, which is the minimum between the available bandwidth of p and that of c_j (line 22). If $\gamma(p)$ is enough to transfer the entire s_k, Escala assigns the full load of migrating s_k to p (lines 23-25). Otherwise, Escala assigns a portion of the load of migrating s_k to p, while pushing s_k to Ω again to assign the remaining load to other paths (lines 26-30).

Algorithm 1 Heuristic algorithm of Escala.

Input and Output: See §V-A and §V-B.
Variable: Ω: stack of the streams to be migrated; P^u: set of paths originated from switch u; $\gamma(p)$: max load that can be assigned to path p

```
 1: $\omega_{c_i}$ = Sort_by_Deadline($\omega_{c_i}$)
 2: if $l_c(c_i) \geq \Theta_{top}$ then                          ▷ $c_i$ is overloaded
 3:     for $s_k \in \omega_{c_i}$ do
 4:         $\Omega$.push($s_k$)                                  ▷ Select $s_k$
 5:         $x^{c_i}(s_k) = 1$
 6:         $l_c(c_i) = l_c(c_i) - r_{c_i}(s_k)$
 7:         if $l_c(c_i) \leq \Theta_{safe}$ then          ▷ Enough streams are selected
 8:             break
 9: else if $l_c(c_i) \leq \Theta_{bottom}$ then                  ▷ $c_i$ is underloaded
10:     for $s_k$ in $\omega_{c_i}$ do
11:         $\Omega$.push($s_k$)                                  ▷ Select $s_k$
12:         $x^{c_i}(s_k) = 1$
13: Locate switch $u$ connected to channel $c_i$
14: $P^u$ = Sort_by_Latency(Get_Paths($u$))
15: while $\Omega$.empty() == False do
16:     $s_k$ = $\Omega$.top(); $\Omega$.pop()
17:     $p$ = $P^u$.begin(); $P^u$.erase($P^u$.begin())
18:     if $p$ == NULL or $t_p(p) + t_c(c_j) > \Phi_{s_k}$ then
19:         return an anomaly to administrators
20:     Locate channel $c_j$ connected to switch $v$
21:     $y_{c_j}^{c_i}(s_k) = 1$, $z_{c_j}^{c_i}(s_k, p) = 1$
22:     $\gamma(p) = \min(b_p(p), \Theta_{top} - l_c(c_j))$
23:     if $r_{c_i}(s_k) \leq \gamma(p)$ then                ▷ $p$ has enough bandwidth
24:         $\alpha_{c_j}^{c_i}(s_k, p) = r_{c_i}(s_k)$         ▷ Migrate the entire $s_k$ on $p$
25:         $l_c(c_j)$ += $r_{c_i}(s_k)$                ▷ Update realtime channel load
26:     else                                       ▷ Bandwidth of $p$ is not enough
27:         $\alpha_{c_j}^{c_i}(s_k, p) = \gamma(p)$              ▷ Migrate a part of $s_k$ on $p$
28:         $r_{c_i}(s_k)$ -= $\gamma(p)$
29:         $\Omega$.push($s_k$)        ▷ Migrate other parts of $s_k$ on other paths
30:         $l_c(c_j)$ += $\gamma(p)$                   ▷ Update realtime channel load
31:     for link $e \in p$ do                          ▷ Update realtime link load
32:         $l_e(e)$ += $\alpha_{c_j}^{c_i}(s_k, p)$
33:     for $p \in P^u$ do                       ▷ Update available path bandwidth
34:         for link $e \in p$ do
35:             $b_p(p) = \min(b_p(p), b_e(e) - l_e(e))$
```

③ Update statistics (lines 31-35). Finally, Algorithm 1 updates the realtime loads of links in p, and then updates the available bandwidth of network paths based on link loads.

Time complexity. Let n denote the number of event streams in c_i; m denote the number of paths in P^u. In ①, Algorithm 1 sorts ω_{c_i}, which complexity is $O(nlog(n))$. Then it enumerates ω_{c_i} to select event streams (lines 2-12), which complexity is $O(n)$. In ②, it sorts P^u (line 14), which complexity is $O(mlog(m))$. Then in ③, it enumerates every path in the network (lines 33-35), which complexity is $O(n \cdot m)$. Thus, the time complexity of Algorithm 1 is $O(nlog(n) + mlog(m) + n \cdot m)$. This is acceptable for online scaling due to two reasons. First, we observe that the number n of event streams is small (less than 1 K in practice). This is because the maximum number of concurrent flows is limited (e.g., a few thousands per second [13]). Second, most networks only have tens of network paths so that m is small (< 100).

VI. Evaluation

In this section, we conduct experiments to evaluate Escala. In each experiment, we present the average after 100 runs. Our experimental results demonstrate that:

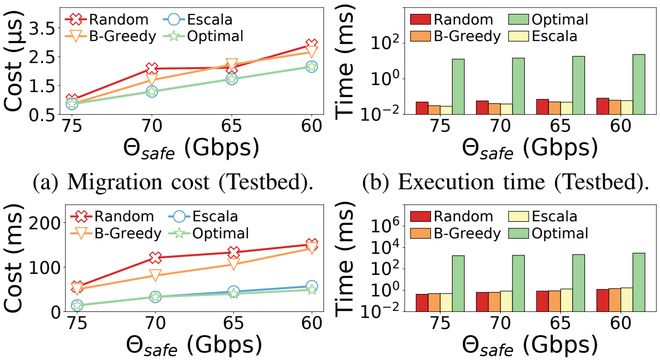
- Escala can effectively mitigate scaling situations. Compared to other algorithms, Escala achieves up to 68% reduction in the cost of migrating event streams while completing its execution within 0.12 ms in all cases (Exp#1-2).
- Escala preserves deadlines in all cases, while other algorithms violate some deadlines due to their sub-optimal scaling decisions and long execution time (Exp#3).
- Escala scales well even with tens of concurrent scaling situations. When 50% control channels in a large-scale network are simultaneously overloaded, Escala can still make near-optimal migration decisions within 8 ms (Exp#4).
- Escala preserves high application-level accuracy via its elastic scaling. In contrast, without Escala, the control channels are easily overloaded, leading to up to 42% accuracy drop (Exp#5). Also, its overhead is acceptable (Exp#6).

A. Experimental Setup

Prototype. We have implemented a prototype of Escala with ~1200 LoC. The prototype comprises three components. First, we implement Escala monitor as a plug-and-play Python module of the network controller, RYU [38]. Second, we build the optimization framework in C++. Third, we implement another RYU module, namely routing coordinator. The coordinator generates routing rules that correctly deliver event streams in the data plane based on scaling decisions. It then configures underlaying switches with routing rules.

Testbed. We build a testbed to evaluate Escala. The testbed comprises a data plane and a control plane. The data plane is a triangle comprising three 32×100 Gbps Tofino switches [14]. The switches are directly connected via 40 Gbps links. Each switch connects to a control plane server via a 100 Gbps control channel. Each server has 36-core Intel(R) Xeon(R) Gold 6240C CPU (2.60 GHz) and 128 GB RAM. We set the deadline Φ_s of collecting events to 5 ms. Unless specified otherwise, we set the thresholds, Θ_{top}, Θ_{bottom}, and Θ_{safe}, to 80 Gbps, 20 Gbps, and 60 Gbps, respectively.

Simulator. We build a simulator to evaluate Escala at scale. We simulate a large-scale wide-area network (WAN), Internet2 [39] (42 switches and 53 links). We set WAN switches and links based on real settings [10, 40]: (1) each switch has 32 100 Gbps ports and connects to the control plane via a 100 Gbps control channel; (2) each switch transfers every packet within 1 μs; (3) the load of each switch is randomly distributed between 20 Gbps and 80 Gbps; (4) the load of each link is randomly distributed between 10 Gbps and 30 Gbps; (5) the latency of each link is randomly distributed between 1 ms and 10 ms; (5) each control channel transfers at least 20 concurrent event streams. We set the deadline Φ_s of collecting events to be randomly distributed between 30 ms and 80 ms.



(a) Migration cost (Testbed). (b) Execution time (Testbed).

(c) Migration cost (Internet2). (d) Execution time (Internet2).

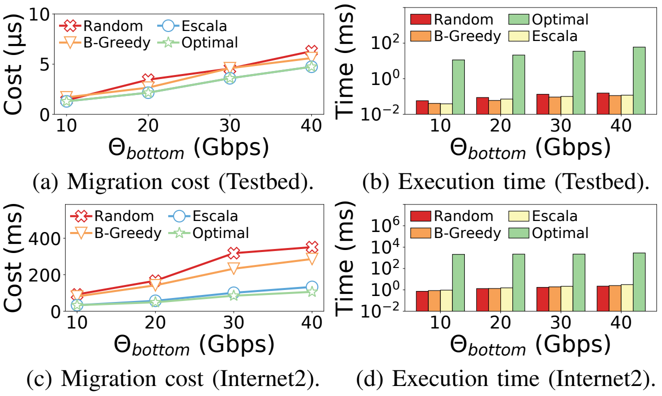
Fig. 2: (Exp#1) Scaling overloaded channels.

Comparison. Since none of existing solutions supports elastic scaling in network measurement, we refer to the flow migration solutions in network function virtualization [26, 27, 28] and design three elastic scaling algorithms, including a random algorithm (Random), a bandwidth-greedy algorithm (B-Greedy), and the optimal algorithm (Optimal). These algorithms use the same method of detecting scaling situations and selecting event streams as Escala. However, Random randomly selects paths to migrate event streams; B-Greedy prioritizes the paths with more available bandwidth resources when selecting paths; Optimal adopts a commodity solver, Gurobi [41], to compute the optimal decisions.

Applications. We build three network management applications, heavy hitter detection [15], superspreader detection [42], and DDoS flow detection [42]. We run these applications on the control plane in our testbed. Moreover, we use P4 [43] to implement five types of sketches, including count-min sketch (CM) [44], count sketch (CS) [45], elastic sketch (ES) [11], MV sketch (MV) [46], and UnivMon (UM) [9]. We deploy these sketches on testbed switches, and allocate these sketches with 10 MB memory that is around the maximum available memory in testbed switches [47]. At runtime, these applications collect sketch buckets from testbed switches and make decisions based on collected buckets.

B. Microbenchmarks

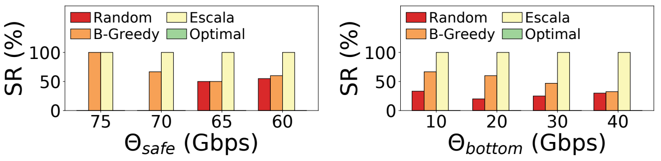
(Exp#1) Scaling overloaded channels. We first use Escala to scale overloaded channels in our testbed. We replay a CAIDA 2018 trace [48] at 100 Gbps to generate a workload. We select one control channel and cause an overload situation by fully loading the channel. We use Escala and



(a) Migration cost (Testbed). (b) Execution time (Testbed).

(c) Migration cost (Internet2). (d) Execution time (Internet2).

Fig. 3: (Exp#2) Scaling underloaded channels in testbed.



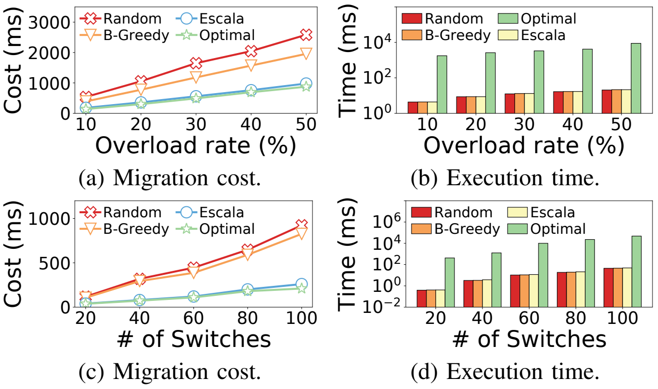
(a) Overload situations. (b) Underload situations.

Fig. 4: (Exp#3) Preserving deadlines.

other algorithms to mitigate the situation, respectively. We vary the safe threshold Θ_{safe} from 75 Gbps to 60 Gbps to simulate different user requirements. Figure 2(a-b) show that Escala outperforms Random and B-Greedy with up to 38% reduction in the cost of migrating event streams (computed by Equation 1). This is because Random and B-Greedy produce sub-optimal solutions, in which paths incur higher cost to migrate event streams. Also, it completes its computation within tens of microseconds, which is orders of magnitude lower than Optimal. Next, we evaluate Escala on the simulated Internet2. Figure 2(c-d) indicate that Escala produces near-optimal decisions in a short time. It reduces the migration cost by up to 75% compared to Random and B-Greedy while reducing the execution time by orders of magnitude compared to Optimal. Its execution completes within a few hundreds of microseconds, which is acceptable for online scaling in large-scale WANs.

(Exp#2) Scaling underloaded channels. We first evaluate Escala in scaling underloaded control channels in our testbed. We randomly select a control channel and make its load below Θ_{bottom}. We vary Θ_{bottom} from 10 Gbps to 40 Gbps. We use Escala to scale in the underloaded channel. In Figure 3(a)-(b), we observe that Escala reduces the migration cost by 37% compared to Random and B-Greedy while converging within 120 μs. Then we evaluate Escala on the simulated Internet2. Figure 3(c)-(d) indicate that compared to Random and B-Greedy, Escala reduces the migration cost by up to 68% while completing its execution with up to 3 ms. In contrast, Optimal requires several seconds to converge, which is inefficient.

(Exp#3) Preserving deadlines. We determine whether Escala preserves the deadlines of collecting event streams. The setup of deadlines is detailed in §VI-A. We use successful ratio (SR), which is the ratio of the number of the event streams that are migrated without violating deadlines to the total number of event streams, as the metric. We measure SR when conducting Exp#1 and Exp#2 on the simulated Internet2. Figure 4 shows that Escala achieves 100% SR, indicating that it completes the migration of all flows without violating any deadlines. This is because both the migration cost and the execution time of Escala are small, such that the sum of the two types of latency does not exceed deadlines. In contrast, other algorithms fail to preserve the deadlines of some event streams due to their sub-optimal migration decisions (in Random and B-Greedy) and long execution time (in Optimal).



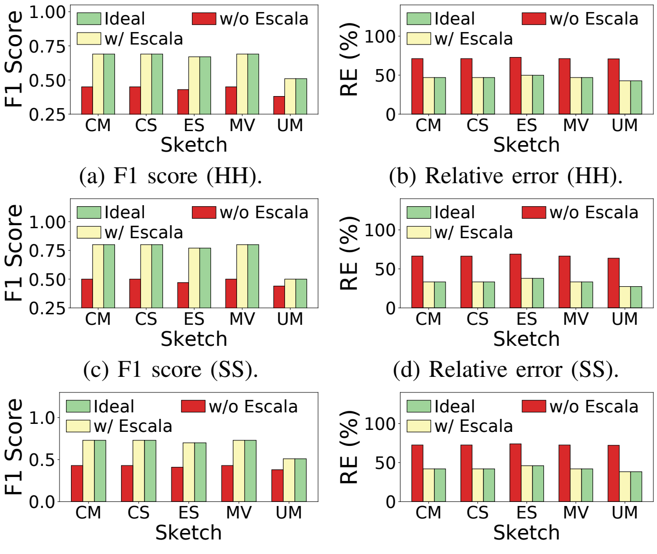
(a) Migration cost. (b) Execution time.

(c) Migration cost. (d) Execution time.

Fig. 5: (Exp#4) Scalability.

(Exp#4) Scalability. We evaluate Escala with several concurrent scaling situations. We simulate the Internet2 topology in our simulator and adopt the same configurations as Exp#1. We manually incur scaling situations by overloading some control channels. We vary the ratio of the number of overloaded channels to the total number of channels (i.e., overload rate) from 10% to 50%. We use Escala and the comparison algorithms to process scaling situations. Figure 5(a) indicates that Escala still produces near-optimal decisions even when 50% channels are overloaded, while the decisions made by Random and B-Greedy incur high migration cost. Note that it presents the sum of migration cost of all event streams. On average, the migration cost of a single event stream is around 8 ms, which is timely and acceptable in the Internet2 topology. Figure 5(b) shows that Escala is able to complete its execution within 22 ms in the worst case. This is acceptable given the large scale of Internet2 with more than 40 control channels. While in most networks such as data center networks, the number of control channels is less than 10. In these cases, the execution time of Escala will not exceed 1 ms (e.g., Figure 2(b)).

Next, we evaluate Escala with large-scale networks in our simulator. We generate random topologies via two steps. First, we vary the number of data plane switches from 20 to 100. Second, each pair of switches are directly connected with a probability of 0.2. Other settings are the same as §VI-A. We repeat Exp#1 on synthetic networks. Figure 5(c)-(d) indicate that Escala makes low-cost decisions in a timely manner. Even in the largest network with 100 switches and around 1 K links, Escala migrates event streams with a near-optimal cost of 260 ms while completing its calculation within 48 ms. Therefore, our results demonstrate that Escala scales well with concurrent scaling situations and complex networks.

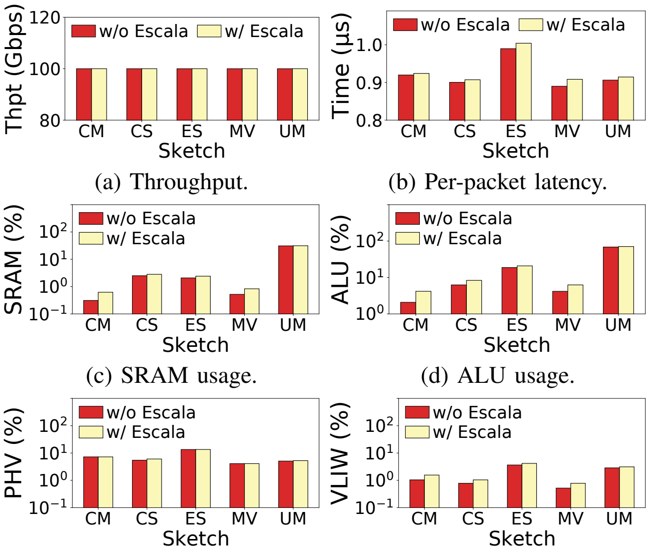


(a) F1 score (HH). (b) Relative error (HH).

(c) F1 score (SS). (d) Relative error (SS).

(e) F1 score (DDoS). (f) Relative error (DDoS).

Fig. 6: (Exp#5) Impact on application-level accuracy.



(a) Throughput. (b) Per-packet latency.

(c) SRAM usage. (d) ALU usage.

(e) PHV usage. (f) VLIW usage.

Fig. 7: (Exp#6) Switch overhead.

C. *Case Study: Sketch-based Measurement*

(Exp#5) Impact on application-level accuracy. In this experiment, we justify if Escala could enhance real measurement applications. We first consider heavy hitter detection. We implement the five sketches on a testbed switch, respectively. We replay our CAIDA trace to generate a workload at 100 Gbps, and direct the workload to traverse the switch. In the control plane, we collect events (i.e., sketch buckets) from the switch via a control channel. We compute heavy hitters based on collected events. The threshold of heavy hitter detection is 10 K packets. We set the collection interval to 0.5 ms [9]. We also compute true heavy hitters in the trace as the baseline. By comparing the measured heavy hitters and true ones, we calculate F1 score and relative error (RE) to quantify application-level accuracy.

We measure the accuracy under two scenarios. First, we measure the accuracy when Escala is not activated. In this case, the rate of collecting events reaches $\frac{10\,\text{MB}}{0.5\,\text{ms}}$=160 Gbps, which overloads the control channel. Second, we activate Escala to handle the situation. Figure 6(a)-(b) show that Escala preserves the original accuracy because it avoids event loss by scaling out the overloaded channel. Thus, all the errors come from the sketch itself. In contrast, without Escala, the control channel is congested, leading to 38% event loss, 33% decrease in F1 score, and 53% increase in relative error.

Next, we study the impact on the accuracy of two security applications, including superspreader (SS) detection and DDoS flow detection [9]. Their thresholds are 0.5% of the total number of IP addresses and that of the total packet number, respectively. Then we compute the true superspreaders and DDoS flows in the CAIDA trace as the baseline. Other settings are the same as Exp#5. Figure 6(c)-(f) show that Escala retains high application-level accuracy as its elastic scaling mitigates scaling situations and thus reduces event loss.

(Exp#6) Switch overhead. Escala needs to configure testbed switches to migrate event streams at runtime. Thus, we measure the overhead of Escala on switches during Exp#5. We first measure normal switch throughput, per-packet processing latency, and switch resource consumption as the baseline. Then we measure the overhead by comparing the results of using Escala with the baseline. Figure 7 indicates that Escala only increases the per-packet processing latency by up to 2.07% without harming throughput. It consumes up to 0.32% SRAM and 2.09% arithmetic and logic units (ALUs), 0.6% packet header vectors (PHVs), and 0.5% very long instruction words (VLIWs), which are small and acceptable.

VII. Conclusion and Future Work

In this paper, we propose Escala that enhances network measurement with elastic scaling of control channels. Escala carefully regulates the loads of control channels by migrating event streams among channels. Thus, it eliminates scaling situations in control channels and avoids event loss. We have implemented Escala on Tofino-based switches. Extensive experiments on a real testbed and large-scale simulation indicate that Escala achieves low-cost and timely elastic scaling.

As in our future work, we plan to enhance Escala with the ability of merging bursty events that are most probably semantically correlated (e.g. congestion) to reduce the load of control channels. We also plan to extend Escala to traditional networks that run distributed routing protocols so as to enhance its scalability in different scenarios.

VIII. Acknowledgement

We thank the anonymous reviewers for their constructive suggestions to this paper. This work is supported by the National Key R&D Program of China (2018YFB1800601), the Science and Technology Development Funds of China (2021ZY1025), the National Natural Science Foundation of China (No. 61902362), and the Key R&D Program of Zhejiang Province (2021C01036, 2022C01085, 2022C01078).

Reference

[1] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, “Turboflow: Information rich flow record generation on commodity switches,” in *EuroSys*, 2018, p. 11.

[2] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, “Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow,” in *ATC*, 2018, pp. 823–835.

[3] X. Chen, Q. Huang, D. Zhang, H. Zhou, and C. Wu, “ApproSync: Approximate state synchronization for programmable networks,” *ICNP*, pp. 1–12, 2020.

[4] C. Xiang, H. Qun, W. Peiqiao, L. Hongyan, C. Yuxin, Z. Dong, Z. Haifeng, and W. Chunming, “Mtp: Avoiding control plane overload with measurement task placement,” in *INFOCOM*, 2021, pp. 1–10.

[5] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, “Scream: Sketch resource allocation for software-defined measurement,” in *CoNEXT*. ACM, 2015, p. 14.

[6] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, “Planck: Millisecond-scale monitoring and control for commodity networks,” in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 407–418.

[7] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, “Dream: dynamic resource allocation for software-defined measurement,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 419–430, 2015.

[8] Y. Li, R. Miao, C. Kim, and M. Yu, “Flowradar: a better netflow for data centers,” in *NSDI*, 2016, pp. 311–324.

[9] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, “One sketch to rule them all: Rethinking network flow monitoring with univmon,” in *SIGCOMM*, 2016, pp. 101–114.

[10] Q. Huang, P. P. Lee, and Y. Bao, “Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference,” in *SIGCOMM*, 2018, pp. 576–590.

[11] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, “Elastic sketch: Adaptive and fast network-wide measurements,” in *SIGCOMM*, 2018, pp. 561–575.

[12] S. Wang, C. Sun, Z. Meng, M. Wang *et al.*, “Martini: Bridging the gap between network measurement and control using switching asics,” in *ICNP*, 2020, pp. 1–12.

[13] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the social network’s (datacenter) network,” in *SIGCOMM*, 2015, pp. 123–137.

[14] Tofino. [Online]. Available: https://www.barefootnetworks.com/technology/#tofino

[15] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, “Heavy-hitter detection entirely in the data plane,” in *SOSR*, 2017, pp. 164–176.

[16] V. Mathew, R. K. Sitaraman, and P. Shenoy, “Energy-aware load balancing in content delivery networks,” in *INFOCOM*, 2012, pp. 954–962.

[17] sflow. [Online]. Available: http://sflow.org/about/index.php

[18] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, “Devoflow: Scaling flow management for high-performance networks,” in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 254–265.

[19] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, “Language-directed hardware design for network performance monitoring,” in *SIGCOMM*, 2017, pp. 85–98.

[20] R. Hark, N. Aerts, D. Hock, N. Richerzhagen, A. Rizk, and R. Steinmetz, “Reducing the monitoring footprint on controllers in software-defined networks,” *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1264–1276, 2018.

[21] C. Jing, M. Zili, G. Yaning, X. Mingwei, and H. Hongxin, “Hiertopo: Towards high-performance and efficient topology optimization for dynamic networks,” in *IWQoS*, 2021, pp. 1–10.

[22] S. Luo, H. Yu, and L. Vanbever, “Swing state: Consistent updates for stateful and programmable data planes,” in *SOSR*, 2017, pp. 115–121.

[23] J. Xing, A. Chen, and T. E. Ng, “Secure state migration in the data plane,” in *SPIN*, 2020, pp. 28–34.

[24] M. He, A. Blenk, W. Kellerer, and S. Schmid, “Toward consistent state management of adaptive programmable networks based on p4,” in *NEAT*, 2019, pp. 29–35.

[25] V. Demianiuk, S. Gorinsky, S. Nikolenko, and K. Kogan, “Robust distributed monitoring of traffic flows,” in *ICNP*, 2019, pp. 1–11.

[26] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, “Opennf: Enabling innovation in network function control,” in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 163–174.

[27] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, “Split/merge: System support for elastic execution in virtual middleboxes,” in *NSDI*, 2013, pp. 227–240.

[28] C. Sun, J. Bi, Z. Meng, X. Zhang, and H. Hu, “Ofm: Optimized flow migration for nfv elasticity control,” in *IWQoS*, 2018.

[29] C. H. Song, P. G. Kannan, B. K. H. Low, and M. C. Chan, “Fcm-sketch: generic network measurements with data plane support,” in *CoNEXT*, 2020, pp. 78–92.

[30] Q. Xiao, Z. Tang, and S. Chen, “Universal online sketch for tracking heavy hitters and estimating moments of data streams,” in *INFOCOM*, 2020, pp. 974–983.

[31] R. Fontugne, C. Pelsser, E. Aben, and R. Bush, “Pinpointing delay and forwarding anomalies using large-scale traceroute measurements,” in *IMC*, 2017, pp. 15–28.

[32] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, and D. Rossi, “Flowmon-dpdk: Parsimonious per-flow software monitoring at line rate,” in *TMA*, 2018, pp. 1–8.

[33] C. Hu, B. Liu, S. Wang, J. Tian, Y. Cheng, and Y. Chen, “Anls: Adaptive non-linear sampling method for accurate flow size measurement,” *IEEE Transactions on Communications*, vol. 60, no. 3, pp. 789–798, 2011.

[34] R. Hark, M. Ghanmi, S. Kar, N. Richerzhagen, A. Rizk, and R. Steinmetz, “Representative measurement point selection to monitor software-defined networks,” in *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*. IEEE, 2018, pp. 511–518.

[35] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[36] LLDP. [Online]. Available: https://www.ieee802.org/1/files/public/docs2002/lldp-protocol-00.pdf

[37] E. C. man Jr, M. Garey, and D. Johnson, “Approximation algorithms for bin packing: A survey,” *Approximation algorithms for NP-hard problems*, pp. 46–93, 1996.

[38] Project Ryu. [Online]. Available: https://osrg.github.io/ryu/

[39] Internet2. [Online]. Available: https://www.glif.is/meetings/2006/plenary/summerhill-internet2.pdf

[40] X. Chen, H. Liu, Q. Huang, P. Wang, D. Zhang, H. Zhou, and C. Wu, “Speed: Resource-efficient and high-performance deployment for data plane programs,” in *ICNP*. IEEE, 2020, pp. 1–12.

[41] Gurobi optimizer. [Online]. Available: http://www.gurobi.com

[42] M. Yu, L. Jose, and R. Miao, “Software defined traffic measurement with opensketch,” in *NSDI*, 2013, pp. 29–42.

[43] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, “P4: Programming protocol-independent packet processors,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[44] G. Cormode and S. Muthukrishnan, “An improved data stream summary: the count-min sketch and its applications,” *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[45] M. Charikar, K. Chen, and M. Farach-Colton, “Finding frequent items in data streams,” in *International Colloquium on Automata, Languages, and Programming*. Springer, 2002, pp. 693–703.

[46] L. Tang, Q. Huang, and P. P. Lee, “Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams,” in *INFOCOM*, 2019, pp. 2026–2034.

[47] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, “Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn,” *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[48] Caida trace. [Online]. Available: http://www.caida.org/data/overview/