ScaleFlux: Efficient Stateful Scaling in NFV

Libin Liu, *Member, IEEE*, Hong Xu, *Senior Member, IEEE*, Zhixiong Niu, Jingzong Li, *Student Member, IEEE*, Wei Zhang, Peng Wang, Jiamin Li, *Student Member, IEEE*, Jason Chun Xue, *Member, IEEE*, and Cong Wang, *Fellow, IEEE*

Abstract—Network function virtualization (NFV) enables elastic scaling to middlebox deployment and management. Therefore, efficient stateful scaling is an important task because operators often need to shift traffic and the associated flow states across VNF instances to deal with time-varying loads. Existing NFV scaling methods, however, typically focus on one aspect of the scaling pipeline and does not offer an end-to-end scaling framework. This article presents ScaleFlux, a complete stateful scaling system that efficiently reduces flow-level latency and achieves near-optimal resource usage. ScaleFlux (1) monitors traffic load for each VNF instance and adopts a queue-based mechanism to detect load burstiness timely, (2) deploys a flow bandwidth predictor to predict flow bandwidth time-series with the ABCNN-LSTM model, and (3) schedules the necessary flow and state migration using the simulated annealing algorithm to achieve both flow-level latency guarantee and resource usage minimization. Testbed evaluation with a five-machine cluster shows that ScaleFlux reduces flow completion time by at least 8.7× for all the workloads and achieves near-optimal CPU usage during scaling.

Index Terms—Network function virtualization, network load detection, flow bandwidth prediction, stateful scaling, service level agreements

1 Introduction

Network function virtualization (NFV) [2], [3] aims to replace hardware middleboxes with virtual software instances running on commodity servers. NFV enables flexible scaling of the virtual instances to better handle time-various network loads. Besides, unlike layer 3 forwarding, many middleboxes such as firewall, proxy, and VPN perform stateful packet processing. Consider a load balancing scenario where a firewall instance is overloaded with traffic, and an additional instance needs to be spawned. Operators need to adjust the flow routing tables to dynamically redistribute packet processing across multiple virtual network function (VNF) instances. They must also move the internal states associated the flows to the new instance so that it can continue processing the traffic without disruption to users.

In addition, many large enterprises and cloud providers report that their networks contain many VNFs performing a wide range of advanced packet processing and their traffic loads change variously over time [4], [5], [6], [7]. These VNFs need dynamic scaling in three aspects. First, VNFs need to scale up to deal with traffic load spikes in order to meet the service level agreements (SLAs) [5], [6], [8]. Second, VNFs need to scale down when load decreases to avoid resource wastage [6], [9], [10]. Third, burstiness is common in data center networks and WANs [4], [6], [7]. VNFs need to timely and efficiently scale to avoid flow-level SLA violation.

As a result, dynamic scaling emerges as an important research issue in NFV. Frameworks such as E2 [10], S6 [11], Metron [12], and FlexNFV [9] dynamically scale VNF service chains to improve their performance. E2 [10] manages VNF placement, service interconnection, and dynamic automatic scaling without operator intervention. To maintain flow affinity, E2 adopts a migration avoidance strategy that only moves flows without states to the new instance. In contrast, S6 [11] performs elastic scaling of stateful network functions. It separates packet processing and their corresponding states by designing a distributed shared state abstraction. By this, S6 simplifies the state migration during scaling. Yet, it requires operators to re-implement VNFs based on its new abstraction. Split/Merge [13] and OpenNF [14] support stateful VNF scaling without the need to re-implement VNFs. They automatically transfer states across VNFs with or without guarantees on packet loss, reordering, and state inconsistency. These work, however, migrates states without latency guarantees, which takes hundreds of

- *Libin Liu is with Zhongguancun Laboratory, Beijing 100190, China. E-mail: liulb@zgclab.edu.cn.*
- *Hong Xu is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong SAR, China. E-mail: hongxu@cuhk.edu.hk.*
- *Zhixiong Niu is with Microsoft Research Asia, Beijing 100080, China. E-mail: zhixiong.niu@microsoft.com.*
- *Jingzong Li, Jiamin Li, Jason Chun Xue, and Cong Wang are with the Department of Computer Science, City University of Hong Kong, Hong Kong SAR, China. E-mail: {jiaminli8-c, jingzong.li}@my.cityu.edu.hk, {jasonxue, congwang}@cityu.edu.hk.*
- *Wei Zhang is with the Shandong Provincial Key Laboratory of Computer Networks, Shandong Computer Science Center (NationalSupercomputer Center in Jinan), Qilu University of Technology (Shandong Academy of Sciences), Jinan, Shandong 250316, China. E-mail: wzhang@sdas.org.*
- *Peng Wang is with Theory Lab, Huawei Hong Kong Research Center, Hong Kong SAR, China. E-mail: wang.peng6@huawei.com.*

Manuscript received 11 January 2022; revised 2 August 2022; accepted 31 August 2022. Date of publication 5 September 2022; date of current version 19 September 2022.
This work was supported in part by the National Natural Science Foundation of China under Grant 61802233, in part by the Pilot Project for Integrated Innovation of Science, Education and Industry of Qilu University of Technology (Shandong Academy of Sciences) under Grant 2020KJC-ZD02, in part by the General Research Fund from Hong Kong Research Grants Council under Grant 11209520, in part by CUHK under Grants 5501329, 5501517, 4937007, and 4937008.
(Corresponding author: Hong Xu.)
Recommended for acceptance by D. Mohaisen.
Digital Object Identifier no. 10.1109/TPDS.2022.3204209

1045-9219 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

milliseconds to complete, generates much overhead in the control plane, and degrades application performance. We find that OpenNF [14] takes more than 100ms to move per-flow states for 1,000 flows.

However, existing work does not migrate the network load exactly at flow level, while guaranteeing per-flow SLA during scaling. Handling network load subtly and achieving per-flow SLA are critical to resource utilization [15], [16] and the quality of user experience [17], [18]. According to our observation on the real traffic from a cloud gateway in Section 2.1, the flow burstiness ratio is high but the number of concurrent bursty flows is limited. This indicates the network load burstiness is commonly caused by a small number of bursty flows. Therefore, this inspires us to design a NFV scaling system that can exactly detect such flows and migrate them, while achieving per-flow SLA.

Thus, in this paper, we propose a dynamic scaling system for NFV, called ScaleFlux. ScaleFlux just needs to maintain a suitable quantity of VNF instances to process the stable network load with high resource utilization. Once it detects the network load exceeds the processing capacity of the instances, it launches scaling at once and predicts the bandwidth for each flow. Then, ScaleFlux can accurately select the flows which lead to the overload issue to migrate, as the flow bandwidth reflects the contribution of each flow to the network load and migrating the "right" flows does solve the overload issue and also reduce the effect to the other flows. Meanwhile, when scheduling the flows to migrate, ScaleFlux provides per-flow SLA guarantee, since flow migration incurs extra latency and flows contributing more to the load burstiness may also be extremely sensitive to the latency, such as short videos [19].

Besides, in order to verify the idea, we build a complete end-to-end scaling system to support the whole NFV scaling process including network load monitoring, scaling scheduling, and flow and state migration. First, to perform scaling timely, a *load monitor surrogate* proactively monitors traffic load of each VNF instance and adopts a queue-based mechanism to detect load changes. Then, in order to migrate the network loads at flow level, ScaleFlux needs to know the future arrival rates of active flows and estimate which ones are likely to contribute more to the future load. We design a *flow bandwidth predictor* that relies on an attention-based CNN-LSTM model to predict flow time-series bandwidth with the historical time-series, which is calculated using the information, such as packet size, number of packets of a flow, and packet timestamps, that we collect in real-time for each flow traversing the VNF. The attention mechanism is used to extract important features from the flow bandwidth time-series and LSTM module is used for time-series prediction. Such a model can avoid the interference of unimportant data and gradient dispersion problems [20].

Once scaling process is triggered, ScaleFlux's *scaling scheduler* runs the simulated annealing algorithm (SAA) to decide which flows should be migrated and where to migrate, in order to minimize VNFs' resource usage while satisfying flow-level latency guarantees. Specially, since existing work separating state management and packet processing [11], [21], [22] does not have open and standard interfaces that are widely used in practice, they are not ready for wide deployment in production to implement various VNFs [23]. Thus, ScaleFlux considers state migration during scaling and SAA takes the state migration overhead into consideration when scheduling. In Section 2.2, we also analyze the overhead of state migration cannot be ignored. Finally, ScaleFlux implements a *flow and state manager* to migrate flows and states. During migration, their packets have to be buffered at the controller to avoid packet loss or reordering at the new instance. ScaleFlux relies on an underlying state management framework such as OpenNF to provide basic state migration services. It adds two new components: a per-flow state migration API for the state manager to enable dedicated flow migration and a filter API that updates flow routing table to redistribute traffic. After finishing migration, ScaleFlux deletes the states of migrated flows from the original instance and completes the scaling process.

We make several contributions in this work.

- We identify the need to consider dynamic scaling in NFV to meet SLA and analyze the overhead of state migration (Section 2).
- We design a VNF scaling framework called ScaleFlux (Section 3) that elastically scales VNF instances to guarantee flow-level latency and minimize resource usage.
- We implement ScaleFlux (Section 4) and present extensive evaluations on a five-machine cluster (Section 5). Experiments with the real workloads show that ScaleFlux substantially minimizes resource costs and provides flow-level latency guarantee, and its components perform efficiently with little overhead. Compared to existing solutions such as OpenNF [14], ScaleFlux reduces flow completion time by at least $8.7\times$ for all the workloads and achieves near-optimal CPU usage.

The rest of the paper is structured as follows. We first introduce the motivation for scaling in NFV to meet SLAs and the overhead of state migration in Section 2. Then, we describe the overall system architecture, detailed component design, as well as the algorithms in Section 3. We discuss the implementation of ScaleFlux in Section 4, and performance evaluation with testbed experiments in Section 5. Finally, we discuss related work in Section 6, and conclude in Section 7.

2 Background and Motivation

In this section, we first introduce the traffic characteristics to motivate the dynamic scaling need to meet the per-flow SLA in Section 2.1. We then introduce the overhead of state migration when VNF scales, to inspire us not only considering flows themselves, but also their states in Section 2.2.

2.1 Need Scaling to Meet SLAs

We collect the traffic from three production cloud gateways of an Internet content provider. The cloud gateways run on commodity servers or programmable switches, and manipulate packet headers and forward them with tunnels using flow tables, e.g., ACL, NAT, and GRE or VxLAN [24], [25], [26]. For each gateway, we collect packet-level traces at different time periods of a typical work day. In the following,

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

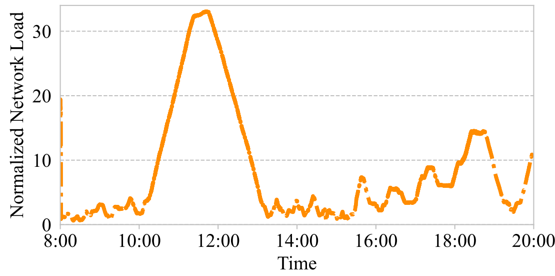


Fig. 1. The time-series for the normalized network load of one cloud gateway. We calculate the overall average from 8:00am to 8:00pm and obtain the normalized network load with respect to the overall average.

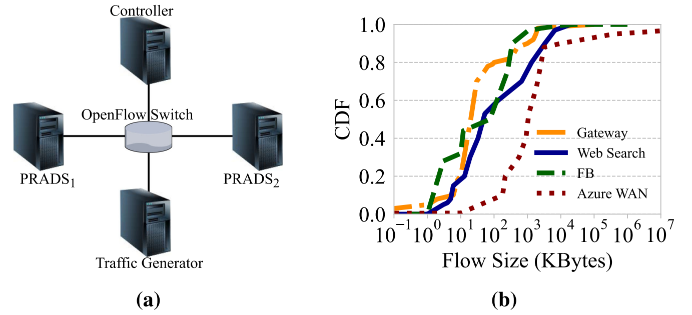


Fig. 3. (a) Testbed topology; (b) Flow size distributions in a cloud gateway, a web search cluster [27], a Facebook cache cluster (FB) [28], and Azure WAN [29]. The network trace for gateway belongs to a cloud gateway cluster from a large cloud service provider.

we describe our key findings from the traces. We only present results of one gateway, since traffic characteristics of the other two demonstrate the same properties.

Network Load Constantly Changes Over Time. We collect the network load every 1ms from 8:00am to 8:00pm of a day and show its time-series in Fig. 1. We summarize the data for each 10-minute period as a data point and calculate the overall average, and Fig. 1 depicts the normalized network load with respect to the overall average. We observe the network load of the gateway changes over time, and the normalized load ranges from 0.16 to 38.43. Besides, we also explore the characteristics of individual flows. We observe that most flows are bursty: the sending rates of *bursty flows* surge quickly and last for a short period. We use the ratio between a flow's peak rate and its average, named *burstiness ratio*, to describe the level of burstiness. We depict the CDFs of burstiness ratio in Fig. 2a. We can see ~80% flows have a ratio larger than 20 and the maximum reaches 80. Moreover, we make a statistical analysis about the distribution of the number of concurrent bursty flows whose burstiness ratio is larger than 10, as shown in Fig. 2b. We find that the number of concurrent bursty flows is limited. The maximum is 49 from the trace of the cloud gateway workload compared to the total number of 500K flows. This also demonstrates that the overall load burstiness is usually because of a small number of bursty flows.

Burstiness Degrades SLAs. We further conduct experiments to show that bursty flows should be handled separately to guarantee their SLAs. Most previous work migrates flows during NFV scaling without considering SLAs [14] or only considering the overall SLAs [1], [9]. They pay little attention to per-flow SLA. Per-flow SLA determines the overall SLA guarantee, but not vice versa. To see this, we quantitatively compare the queue size and processing latency for processing bursty and stable flows by a VNF, respectively. Our testbed servers are connected as shown in Fig. 3a. We use one server as the sender, another as the receiver, and a third one with one CPU core running PRADS [30] as a VNF. We generate stable and bursty flows at the sender. Note that the average rate of the stable flows is 5 times higher than that of bursty flows.

Fig. 4 shows that, for stable flows and the stable period of bursty flows, queue size and packet latency are both very small. Yet, at the peak period of the bursty flows, they are much higher than the normal. This is because CPU resources are reserved for a VNF instance to provide fixed processing capacity and the instantaneous arrival rate exceeds the VNF's processing capacity and unprocessed packets are buffered at the queue.

Scaling Counteracts Burstiness. Continuing with the same setup, we vary the number of PRADS instances for processing the bursty flows, and measure the effect on packet loss rates and latency. Besides, to balance the load, flows are randomly hashed to the instances. As shown in Fig. 5, as the number of instances increases, packet loss rate and latency are reduced. We observe that four instances are needed for bursty flows to avoid packet loss and achieve the same average latency with stable flows, which only entails one instance. The results indicate that bursty flows need more instances in parallel to ensure satisfactory SLAs. Clearly, we should scale the VNF instances dynamically according to

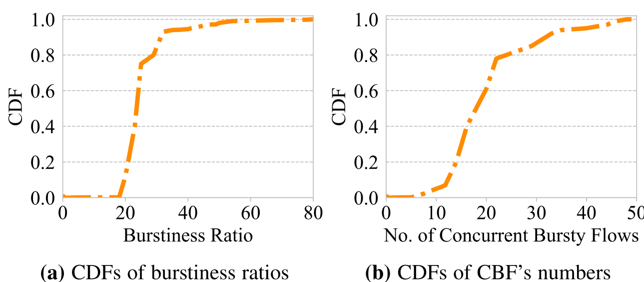


Fig. 2. Traffic characteristics of the cloud gateway. The burstiness ratio is calculated by a flow's peak sending rate regarding to its average and the full name of CBF is concurrent bursty flow.

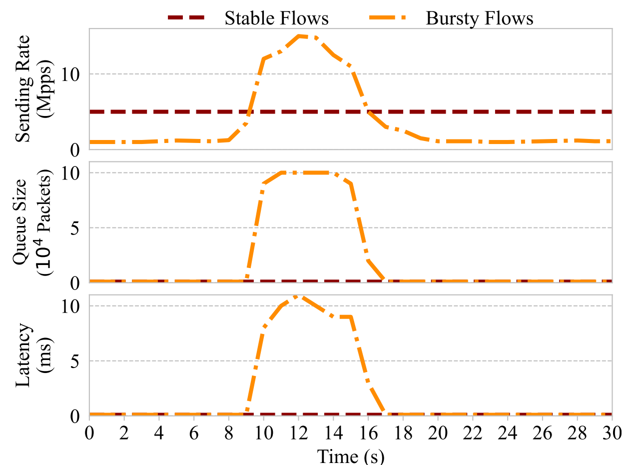


Fig. 4. Sending rate, queue size, and latency for stable flows and bursty flows.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

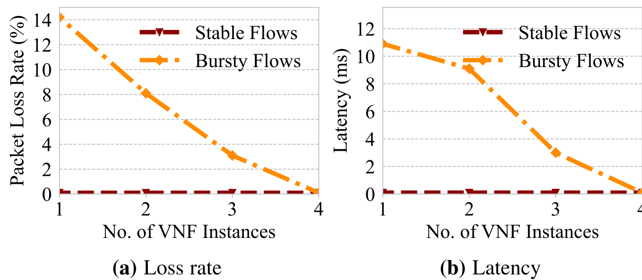


Fig. 5. Packet loss rate and latency against different numbers of VNF instances for stable flows and bursty flows, respectively.

the instantaneous traffic load of bursty flows in order to avoid resource over-utilization or under-utilization while guaranteeing SLAs.

2.2 Overhead of State Migration

Most VNFs perform stateful packet processing. Thus, when migrating the flows we also need to migrate their associated states from original VNF instance to the new one, which incurs non-negligible overhead. To understand this overhead, we deploy a five-machine testbed as shown in Fig. 3a and measure the time used to migrate states using the network workloads whose CDFs of flow sizes are shown in Fig. 3b. They are from a cloud gateway cluster as mentioned before, a web search cluster [27], a Facebook cache cluster (FB) [28], and Azure WAN [29], respectively. Flows arrive according to a Poisson process with an average load of 1.

Our observation is that state migration incurs downtime on the order of $O(100)$ms [10], [14], which is significantly larger than the flow completion time of many flows, especially the mice ones. This is because migrating states is a control-plane action where complex optimization mechanisms and serialization/deserialization have to be in place to provide critical performance guarantees such as loss-free and order-preserving for NFV applications. To see this, we deploy OpenNF [14], state-of-the-art state migration system, on the testbed (more details of the testbed in Section 5.1). We use five physical machines instead of VMs for maximum performance. We use two PRADS asset monitor instances [30] ($PRADS_1$ and $PRADS_2$), which are OpenNF-enabled. Initially all traffic is sent to $PRADS_1$. After it has created per-flow states for 2,000 flows, we move half of the flows and their states to $PRADS_2$ for load balancing. Fig. 6a shows that even without any performance guarantees, OpenNF takes at least 268ms to transfer 1,000 states. When applications demand loss-free and/or order-preserving guarantees, the migration time is beyond 400ms even with optimizations.

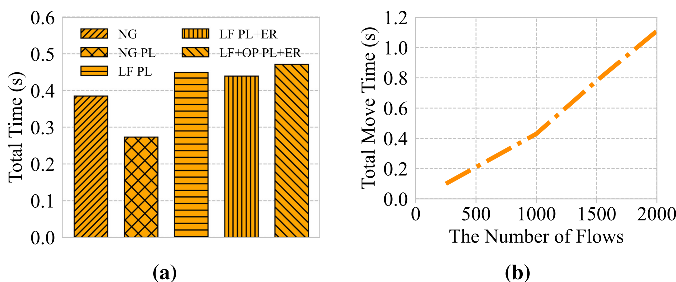


Fig. 6. (a) Migration time with no guarantees (NG), loss-free (LF), and loss-free and order-preserving (LF+OP) with and without parallelizing (PL) and early-release (ER) optimizations, using flow size distributions of Fig. 3b; (b) Migration time when moving different numbers of flows with loss-free and order-preserving guarantees, and parallelizing and early-release optimizations in OpenNF.

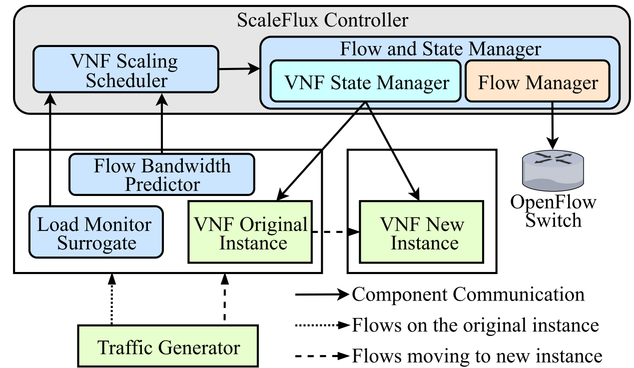


Fig. 7. The system overview of ScaleFlux.

This motivates us to take both the flows to migrate and their states into consideration when scaling VNF instances. We should select the flows which can reduce the VNF load bursty efficiently while reducing the downtime for state migration. This is because state migration time cannot be ignored for all the flows, especially for mice flows, which may finish the transmission long before state migration ends. On the one hand, migrating large flows or bursty flows can quickly deal with traffic load burstiness and avoid SLA violation, especially for an individual bursty flow. However, migrating other flows (e.g., mice flows) will significantly harm their SLAs and cannot deal with load burstiness of VNFs. We do not focus on the consistency issues that arise when global or multi-flow state is needed on the new VNF instance, which is addressed in some existing work [14]. Our experiments only migrate per-flow state.

The benefits of carefully selecting flows to migrate is three-fold. First, it can handle the flows' migration downtime. We repeat the OpenNF migration experiments with varying number of states and find that migration downtime increases with number of states as shown in Fig. 6b.[1] Thus, migrating fewer states can greatly cut the downtime and eliminate the extra latency for flows. Second, it can help avoid SLA violation for individual flows, especially bursty flows, as shown in Fig. 5. Third, it also helps minimize migration overhead at both the controller and OpenFlow switches. Because the controller has fewer packets to buffer and send messages for. Additionally, fewer forwarding rules need to be updated at OpenFlow switches to adjust routing of flows, further streamlining the entire migration process with flows and states.

3 Design

We now present the design of ScaleFlux in this section. We start by presenting the system overview in Section 3.1. We then explain the load monitor surrogate in Section 3.2, and flow bandwidth predictor in Section 3.3. Lastly, we discuss

1. This is likely due to the increased complexity of providing loss-free and order-preserving guarantees with more packets.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

the scaling scheduler to achieve both flow-level latency guarantee and resource efficiency in Section 3.4.

3.1 Overview

ScaleFlux is an automatic stateful scaling system in NFV. Fig. 7 shows the system overview of ScaleFlux. It includes four key components: load monitor surrogate, flow bandwidth predictor, scaling scheduler, and flow and state manager. The *load monitor surrogate* monitors queue size of each VNF instance and detects network load burstiness in real time. When the network load exceeds a predefined threshold, VNF scaling is triggered, and then the *flow bandwidth predictor* runs the attention-based CNN-LSTM model to forecast the bandwidth time-series for individual flows and calculates their future arrival rates. Next, the *scaling scheduler* determines the flows to migrate with flow-level latency guarantee and resource efficiency. Finally, ScaleFlux uses the *flow and state manager* to finish the migration of designated flows and their corresponding states.

We explain the key components of ScaleFlux below.

Load Monitor Surrogate. ScaleFlux employs a load monitor surrogate for each VNF instance to continuously monitor the overall network load in real time. The load monitor surrogate monitors the queue size and compares it against the predefined threshold. Once it finds the queue size exceeds the threshold, it will report to scaling scheduler to launch the scaling process.

Flow Bandwidth Predictor. Flow bandwidth predictor adopts an attention-based CNN-LSTM (ABCNN-LSTM) model to forecast time-series bandwidth for each flow and then calculates their arrival rates for the future epoch accordingly. It continuously collects flows' bandwidth information and makes it as the input of ABCNN-LSTM at each VNF instance. The prediction results are used for scaling scheduling, as they can reflect the contribution of each flow to the total network load. Besides, ScaleFlux uses the information to update the model offline and adapts to the characteristics of bandwidth time-series over time.

Scaling Scheduler. The scaling scheduler in ScaleFlux runs the simulated annealing algorithm (SAA) to achieve both flow-level latency guarantee and resource efficiency. It listens to the scaling launching event from load monitor surrogate and obtains flows' future arrival rates from flow bandwidth predictor. Then, it determines which flows and states to migrate to the new instances, and invokes flow and state manager to finish migration.

Flow and State Manager. Flow and state manager is waiting for the commands on flow and state migration and subsequently migrates the ones specified by scaling scheduler. Once invoked, it adjusts the flow routing table at switches, iteratively moves the states of selected flows to new VNF instances, and deletes the states at the original instance and finishes migration process.

3.2 Load Monitor Surrogate

As shown in Fig. 8a, the load monitor surrogate leverages a process to proactively monitor and detect network load changes for each VNF instance. When the software queue size exceeds a pre-defined threshold K, some flows processed by the VNF instance need to be migrated to new instances to balance load and guarantee flows' SLAs. Load monitor surrogate leverages counters to accurately monitor the queue size, which can be software counters or hardware counters within a commodity NIC.

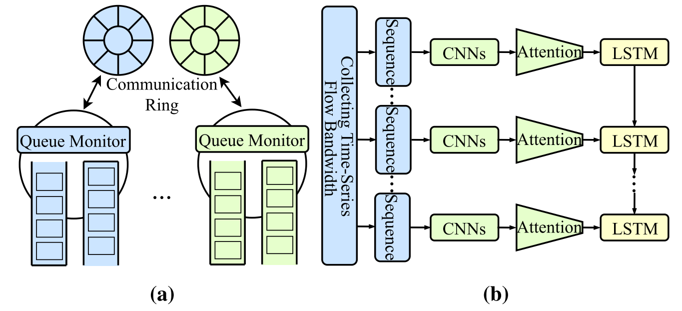


Fig. 8. (a) Ring buffer for queue-based load detection in load monitor surrogate; (b) The architecture of ABCNN-LSTM model.

Queue Size Threshold. Most practical NFV systems use high performance packet IO tools, e.g., DPDK [31] and netmap [32] for traffic processing. Within these systems, the software and hardware communicate with each other via a ring buffer. That is, the hardware directly places packets into the ring buffer to send them to the software. Consequently, when the VNF processing speed cannot catch up with the traffic arrival rate during network load rising, packets will be buffered and the packet queue will build up in the ring buffer over time. Therefore, a sudden increasement of packet queue size is a clear signal for the existence of traffic load burstiness.

One important problem here is how to set the threshold K to accurately detect network load burstiness and thus trigger VNF scaling timely. Inspired by the idea on setting the headroom size in PFC [33], [34], we set K accordingly. For PFC, the headroom should be large enough to store the packets received by a switch between the time point when the PFC pause message is sent and the one when the message takes effect. Similarly, a headroom should be reserved to store the packets received by the ring buffer between the time point when VNF scaling is triggered and the one when flows which are responsible for the load bursty are successfully started migration. Following this principle, K is set as:

$$K = Q_{cap} - (T_{idt} + T_{mgt}) \times M_{rate}, \tag{1}$$

where Q_{cap} represents the overall queue capacity for each VNF instance, T_{idt} is the time required for identifying the flows to migrate, T_{mgt} is the time taken for launching the flow and state migration process, and M_{rate} is the flow and state migration rate. We will discuss the concrete setups for Q_{cap}, T_{idt}, T_{mgt}, and M_{rate} in Section 4. With this queue-based mechanism, the load monitor surrogate can work efficiently to detect load burstiness while avoiding packet drops before migration.

3.3 Flow Bandwidth Predictor

To accurately predict flow bandwidth, we employ an attention mechanism to dynamically use the time-series features. The attention network in ScaleFlux is designed to determine the weights for the features since no prior information is available for the attention network to specify weights, we adopt an unsupervised model where the inputs to the

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

model are all the features. We present the architecture of the unsupervised ABCNN-LSTM model in Fig. 8b, which includes input module, CNN module, attention module, and LSTM module. Flow future arrival rate is the average of predicted bandwidth time-series. It is obtained as follows. First, we preprocess the historical bandwidth time-series for each flow as input. Then, the model uses CNN [35] and attention modules to select the important features to focus on. Next, the model adopts LSTM unit to predict bandwidth time-series with the output of attention module. Finally, we calculate their average as flow's future arrival rate.

Flow Monitor. Flow bandwidth predictor relies on a *flow monitor* to collect information of active flows at each VNF instance. Here we consider active to mean existing within the last 1000ms time window. The monitor maintains a *flow information table* (FIT), where each entry stores the flow ID hashed from five-tuples, its current size, packet timestamps, and the list of calculated bandwidths. There are three operations associated with the FIT: insertion, update, and eviction.

Insertion. When a packet arrives at the VNF instance, if the flow is not present in the FIT, a new entry with the flow ID and timestamp is added.

Update. When a packet arrives and its flow is present in the table, the size and timestamp of the flow is updated. Besides, every 5ms, flows' bandwidths are calculated and the new values are added to the list.

Eviction. Every second, an eviction pointer goes through the FIT to evict entries whose timestamps are not within the last time window. This helps maintain the FIT size in a reasonable level. We have varies time window values from 20ms to 2000ms to evict flow entries in Section 5.6. We choose 1000ms because it is able to cover all active flows that indeed cause overload for VNF instance. Of course, ScaleFlux can also adjust the time window size according to the workloads.

The flow monitor adds delay to VNF's packet processing. This overhead can be minimized by implementing it using high-performance packet I/O frameworks such as DPDK [36] or netmap [32] with microsecond level delay. Besides, in ScaleFlux, we convert the collected flow information into corresponding bandwidth time series in milliseconds.

Attention-Based CNN Unit. Various attention-based CNN models have been widely used to help improve the performance by paying attention to important features [20]. The formal definition of the attention mechanism is given as follows,

$$e_{ij} = a(s_{i-1}, h_j), \tag{2}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T^x} \exp(e_{ik})}, \tag{3}$$

$$c_i = \sum\nolimits_{j=1}^{T^x} \alpha_{ij} h_j, \tag{4}$$

where s_{i-1} is the matching feature vector based on current task, h_j is the feature vector of a time point in the time-series, e_{ij} is the unnormalized attention score, and α_{ij} is the normalized attention score. Besides, c_i is the feature of the current time point calculated based on the attention score α_{ij} and the feature sequence h_j.

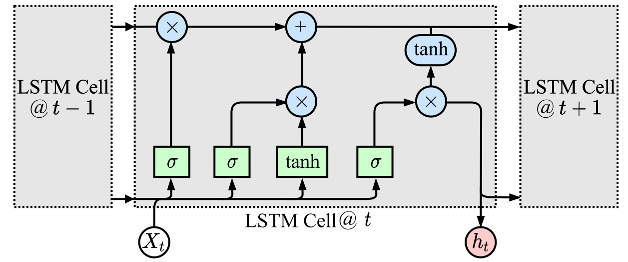


Fig. 9. Long-short-term-memory network.

In addition, the CNN unit consists of multiple layers, each of which includes a batch normalization layer, a convolution layer, a pooling layer, and a non-linear layer. The CNN module achieves sampling aggregation and extracts more features using pooling layer and convolution layer, respectively. It outputs m feature sequences of length n, and the size can be expressed as $n \times m$. In order to obtain the important time-series features, we combine attention module and CNN module in a serial fashion. Attention module consists of feature aggregation and scale restoration components. The feature aggregation component leverages multiple convolutions and pooling layers to extract important features from the sequence and uses a convolution kernel to extract the relationship. And the scale restoration component restores the important features to size $n \times m$, which corresponds to the size of the CNN module's output.

The attention mechanism helps the model classify important features from input sequences and makes it obtain more comprehensive contextual information. In addition, the attention module can help prevent the interference of unimportant features and improve the model's prediction accuracy.

LSTM Unit. In ScaleFlux, we use long-short-term-memory (LSTM) [37] network, to support accurately predict flows' time-series bandwidth. As shown in Fig. 9, LSTM uses a "gate" structure to delete or add information to the state of the cell. The "gate" structure is a method of selectively filtering information. LSTM cells include forget gates f_t, input gates i_t, and output gates o_t. An LSTM cell computes the following:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}), \tag{5}$$

$$f_t = \sigma(W_{if} + b_{if} + W_{hf}h_{(t-1)} + b_{hf}), \tag{6}$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{ho}), \tag{7}$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}), \tag{8}$$

$$c_t = f_t \star c_{(t-1)} + i_t \star g_t, \tag{9}$$

$$h_t = o_t \star \tanh(c_t). \tag{10}$$

Here h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. $W_{\dots}$ are the weights of the respective DNNs in the rectangle nodes in the computation graph in Fig. 9, and $b_{\dots}$ are their respective biases. σ is the sigmoid function, and $\star$ is the Hadamard product.

Flow Future Arrival Rate. Based on the predictions from ABCNN-LSTM, we then calculate the future arrival rate for each flow. Assuming the prediction result for each flow f is $B_T^f = \{b_{t_1}, b_{t_2}, b_{t_3}, \dots, b_{t_n}\}$, thus flow f's arrival rate within

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

the time period T is,

$$\lambda_f = \frac{\sum_{i=t_1}^{t_n} b_i t}{T}, \tag{11}$$

where λ_f represents the predicted arrival rate of the flow f within T, and b_i represents the predicted bandwidth at each time point i. Besides, t is a constant and represents the corresponding interval for updating bandwidth measurement.

3.4 Scaling Scheduler

ScaleFlux controller consists of two components: scaling scheduler and flow and state manager. The scaling scheduler listens to the messages from load monitor surrogate and flow bandwidth predictor, determines which flows to migrate and which instances to move to, and then invokes flow and state manager to take actions to finish migration. It optimizes to minimize resource usages of VNF instances, as well as providing flow-level latency guarantee. The flow and state manager manages both the flow routing table at switches and flow state migration between VNF instances with the APIs which we describe in Section 4.

Note that during state migration, the original VNF instance stops establishing new states for flows it has not seen yet. All packets that hit the original instance but do not have any matching state are forwarded to the new instance which processes them normally. For example for a persistent TCP connection that does not always have traffic, ScaleFlux would not regard it as a new flow, which guarantees the correctness of VNFs.

Scaling Problem Formulation. We begin with a formulation regarding to resource cost of VNF instances and flow-level latency constraints. The number of instances for VNF F is denoted by n, and these instances perform the same packet processing. The queue capacity for each VNF instance i is denoted by q_i^{cap}. Let $Q^{cap} = (q_1^{cap}, .., q_n^{cap})$ be the queue capacity vector for all instances. Each VNF instance is allocated a fixed bunch of resources (e.g., <2 CPU cores, 10GB Mem>) and monopolizes them. This is practical since it avoids resource contention [38], [39], [40] and provides high performance. To simplify it, we use r to indicate the fixed CPU resource for each instance. Thus, the total CPU resource cost for VNF F is,

$$c = nr. \tag{12}$$

Then, the number of flows processed by VNF instance i is denoted by h_i. Thus, the total number of flows for VNF F is,

$$H = \sum_{i=0}^{n} h_i. \tag{13}$$

The maximum latency that flow j can accept is denoted by l_j^{max}. Let $L^{max} = (l_1^{max}, \ldots, l_H^{max})$ be the maximum latency vector for all the flows. We assume packets of flows arrive according to a Poisson process and use the average bandwidth of flow j to represent its packet arrival rate λ_j within the time interval T. The vector $\lambda = (\lambda_1, \ldots, \lambda_H)$ represents the arrival rates for all the flows. Besides, the maximum downtime that flow j accepts is denoted by t_j^d during migration. We use the vector $T^d = (t_1^d, \ldots, t_H^d)$ to indicate all flows' downtime constraints.

We then analyze the service latency of VNF instance i as follows. The total packet arrival rate to VNF instance i,

$$\beta_i = \sum_{j=0}^{n_i} \lambda_j. \tag{14}$$

As a result, the queueing latency of VNF instance i is,

$$l_i^q = \begin{cases} \frac{\beta_i - S_i}{S_i}, & \beta\text{i} \geq \text{Si}, \\ 0, & \text{Otherwise.} \end{cases} \tag{15}$$

The average packet processing latency of instance i is,

$$l_i^p = \frac{1}{S_i}. \tag{16}$$

The service latency of instance i includes queueing latency and processing latency. Thus, the service latency experienced by flow j at instance i is,

$$l_{ij} = \begin{cases} l_i^q + l_i^p, & \text{flow j is processed by VNF instance i,} \\ 0, & \text{Otherwise.} \end{cases} \tag{17}$$

The queue size for each instance i within the time interval T is,

$$q_i = \begin{cases} (\beta_i - S_i)T, & \beta\text{i} \geq \text{Si}, \\ 0, & \text{Otherwise.} \end{cases} \tag{18}$$

We use a vector $Q_i = (q_1, \ldots, q_n)$ to indicate the queue sizes of all instances. During migration, the packets of all the migrated flows will be buffered until all their states are finished migration. We use t^{mgt} to indicate the migration time for each flow, which is a constant and only depends on the bandwidth capacity between VNF instances. We use the vector M_{ij} to indicate the flows to migrate from VNF instance i, where,

$$M_{ij} = \begin{cases} 1, & \text{flow j will be migrated,} \\ 0, & \text{Otherwise,} \end{cases} \tag{19}$$

and the total migration time for flow j at instance i is,

$$T_{ij} = M_{ij} \star t^{mgt}. \tag{20}$$

In addition, we adopt the matrix $B_{f\times n}$ to represent whether the flow $j \in f$ is processed by instance i. Thus,

$$B_{ji} = \begin{cases} 1 & \text{flow j is processed by VNF instance i,} \\ 0 & \text{Otherwise.} \end{cases} \tag{21}$$

The scaling problem can be formulated as the following:

$$\min \quad c \tag{22}$$
$$\text{s.t.} \quad B_{ji} \star l_{ij} \leq l^{max}, \tag{23}$$
$$Q_i \leq Q^{cap}, \tag{24}$$
$$B_{ji} \star T_{ij} \leq T^d, \tag{25}$$
$$\sum_{j=0}^{f} M_{ij} \leq h_i, \tag{26}$$
$$n \in \mathbb{Z}_{\geq 0}. \tag{27}$$

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
The Notations Used in Algorithm 1

Notation	Explanation
T_{max}	The initial temperature
T_{min}	The stop temperature
t	The variable used for the loop
n_t	The number of VNF instances at time t
$\triangle R$	The difference between $n_{t+1}r$ and n_tr
n_{new}	The number of newly instantiated VNF instances

Note that we do not explain the notations already used in the problem formulation.

We formulate the scaling scheduling problem as a nonlinear integer programming problem. The constraints are nonlinear, which makes the problem computationally expensive to solve. Besides, the scheduling task needs to consider some more practical issues, such as VNF instance initialization, flow routing table update, packet buffering during migration, etc.. Therefore, in order to deploy ScaleFlux practically, we design a heuristic algorithm to solve the problem to minimize resource usage while providing flow-level latency guarantee.

Scaling Scheduling Algorithm. We utilize the simulated annealing algorithm (SAA) to solve the scaling scheduling problem as it has demonstrated its efficiency in existing work [41], [42]. SAA's main logic is stated in Algorithm 1 and the notations used are shown in Table 1. We use the function `SAA()` to implement the main logic of the algorithm. First, we initialize the number of VNF instances using `Init()` function. Then, we call a function `ProduceNew()` to produce the next potential number of VNF instances. Sequentially, we calculate the different value between the adjacent resource usages with corresponding VNF instances. If the difference value $\triangle R$ is larger than 0, the number of VNF instances n is updated. Otherwise, it is updated with the possibility $\exp(\frac{\triangle R}{T}) \geq$ `random(0, 1)`. Next, we use `Check()` to see if we can find a M_{ij} from all possible solutions that satisfies constraints defined in Eqs. (23), (24), (25), and (26). If yes, the resource usage returns; otherwise, an infinity. Until the temperature T reaches the designated minimum value T_{min}, `SAA()` returns the number (n_{new}) of new VNF instances to instantiate and flows (M_{ij}) to move.

4 Implementation

We implement the prototype of ScaleFlux with C and Python codes. The load monitor surrogate and flow bandwidth predictor communicate with the scaling scheduler via RPC. We develop a library for the five primary components: load monitor surrogate, flow monitor, ABCNN-LSTM, scaling scheduler, and flow and state manager. ScaleFlux can use hardware counters within the commodity NIC to collect flow information, such as five tuples of packets, packet numbers, packet size, and packet timestamp, and thus offload FIT into the NIC.

Load Monitor Surrogate. ScaleFlux leverages the queue size as an indicator to monitor network load changes. The queue size of each ring buffer can be calculated by polling the size indicator from corresponding NIC pointers. We jointly use the software consumer index pointer (`rq_ci`) and the hardware producer index pointer (`rq_pi`) to calculate the queue size as Q_{cap} (`rq_ci - rq_pi`). Therefore, we implement a NIC pointer reading operation within the kernel's packet receiving logic. To achieve real-time load detection, ScaleFlux reads the pointers upon each packet arrival. Of course, ScaleFlux can adopt virtual tables to record the mappings between flows and VNF instances dynamically [43] and assign a dedicated queue to a VNF instance. According to the virtual tables, ScaleFlux can distribute flows to the dedicated queues. Therefore, ScaleFlux can support multiple VNF instances simultaneously on a physical server. Besides, both T_{idt} and T_{mgt} can be obtained experimentally. T_{idt} is the time period for running SAA, thus we set it as the maximum SAA execution time measured in Section 5.4, which is 4.37ms. Similarly, we set T_{mgt} as 0.08ms in the experiments.

Algorithm 1. SAA for Scaling Scheduling

1: **Input**: Q^{cap}, r, λ, h_i, L^{max}, T^d, T_{max}, T_{min};
2: **Output**: n_{new}, M_{ij};
3: **function** `SAA()`
4: $t = T_{max}$;
5: `Init()`;
6: **for** $t \geq T_{min}$ **do**
7: n_{t+1} = `ProduceNew()`;
8: The resource usage when $n_{new} = n_t$: n_tr;
9: $\triangle R = (n_{t+1} - n_t)r$;
10: **if** $\triangle$R ≥ 0 **then**
11: $n_{new} = n_{t+1}$;
12: **else if** $\exp(\frac{\triangle R}{t}) \leq$ `random(0, 1)` **then**
13: $n_{new} = n_{t+1}$;
14: **if** `Check(n)` != ∞ **then**
15: Update the optimal solution n_{new};
16: $t = t - 1$; return n_{new}, M_{ij};
17: **function** `Init()`
18: The scheduling result at time t: n_t;
19: n_t = `randint()`;
20: **function** `ProduceNew()`
21: i = `randint(1, n)`, b = `randint(-1, 1)`;
22: **if** $n_i + b \leq 0$ **then**
23: $n_i = n_i + b$;
24: return n_i;
25: **function** `Check()`
26: **if** *finding a* M_{ij} *which satisfies all constraints* **then**
27: return $n_{new}r$, M_{ij};
28: **else**
29: return ∞;

Flow Monitor. The flow monitor in flow bandwidth predictor is responsible for managing FIT in the software and hardware, to collect and update flow information. To implement flow monitor using commodity NIC, we leverage the `rte_flow_create` and `rte_flow_destroy` of the `rte_flow` API in the commodity NIC to insert and replace FIT entry. We do not implement it using switch [44], [45], [46], because it takes more time to detect the load burstiness and forward the signal packets to scaling scheduler and monitors excessive flows for other VNFs.

ABCNN-LSTM. We implement the ABCNN-LSTM model on top of PyTorch 1.5.1 and CUDA 9.2 with Python 3.8.0.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

We set the learning rate as 0.001 and the mini-batch size as 256. To make it practical, we train the model using a federated fashion with the generated network traces from the four network workloads, which will be introduced shortly in Section 5.1. Besides, for the model training phrase, we randomly select 70% flows from each network trace to train the ABCNN-LSTM model. All the network traces we use have various network loads ranging from 1Gbps to 10Gbps. This suits for the NFV scenarios where the network load changes over time and the traffic that VNF processes is from various applications. In addition, to determine the best tradeoff between model accuracy and training efficiency, we employ 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.9, 1, 10 as the thresholds of gradient and experimentally select 0.25 as the final one as illustrated in Section 5.3.

Flow and State Manager. ScaleFlux provides a new API call `moveFlows`for VNF scaling scheduler to enable selective flow and state migration. It transfers both the state and input (i.e., traffic) for a set of flows from one VNF instance to another one. Its syntax is:

```
moveFlows(src, dst, scope, properties).
```

The implementation extends OpenNF's `move` method. The `scope` argument specifies which class(es) of state (per-flow and/or multi-flow) to move. We only consider per-flow in this paper. The `properties` argument defines whether the move should be loss-free and order-preserving [14].

When the VNF scaling scheduler issues `moveFlows`, it sends the selected flows, corresponding states, and designated new VNF instances to flow and state manager. Then, the flow and state manager receives the call and communicates with the library patch inside VNF original instance. This is not available in OpenNF whose original API only allows applications to specify which classes of flows should be moved, such as all flows, or flows destined to port 80. This is done through the filter API described below.

The filter API between the controller and the VNF instances consists of three functions: `FilterChannel`, `SetFilter`, and `UpdateFilter`. When the flow and state manager receives `moveFlows` call, it invokes the `FilterChannel` function to send a flow moving message to the VNF instance. The VNF instance then uses the `SetFilter` function to package the five-tuples of selected flows, which is a dictionary specifying values for five-tuples in OpenNF. The flow and state manager uses the matched filters to configure the corresponding `moveFlows` operation. For each matched filter, it invokes the OpenNF `getPerflow` function and passes the filter as the input to get the per-flow state pertaining to the flow. For a move without guarantees, the state manager calls `putPerflow` on the destination VNF instance and `delPerflow` on the original VNF instance to complete per-flow state transfer. Meanwhile, the flow manager uses `UpdateFilter` to make the OpenVSwitch to update the flow tables correspondingly. In detail, in our new filter APIs, we rewrite the `RuleSetup` and `Operation` functions, which can first build connection with library patched in VNF instances. The `setfilter` function tells `getPerflow` function the filters and make it export associated per-flow state from VNF. In this way, we implement to just get per-flow state for flows we selected.

5 Evaluation

In this section, we evaluate ScaleFlux through a series of testbed experiments on a five-machine cluster using traffic traces from four industrial workloads. Our evaluation seeks to answer the following questions:

- *How well does ScaleFlux's load monitor surrogate perform (Section 5.2)?* Load monitor surrogate can detect traffic load burstiness efficiently with at least 0.89ms for 50Mpps peak rate, and at most 10.81ms for 10Mpps.
- *How accurately does flow bandwidth predictor perform (Section 5.3)?* Compared against CNN-LSTM, LSTM, GRU, SVR, and ARIMA, ABCNN-LSTM can predict flow bandwidth more accurately with at least 34.04% better accuracy on average across the four workloads.
- *How efficiently does VNF scaling scheduler work (Section 5.4)?* Across the four workloads, the VNF scaling scheduler achieves at least 217% average latency reduction and at most 402%, and achieves near-optimal CPU resource usage during scaling.
- *How much performance benefit can ScaleFlux provide compared to OpenNF (Section 5.5)?* ScaleFlux reduces the downtime by at least 60.12%, 72.90% on average, and up to 80%, compared to OpenNF. It saves at least 80.50% buffer usage and 83.48% on average, and the saving can be up to 87.87%. Besides, ScaleFlux reduces extra end-to-end latency to migrated flows by at least 88.51%, 93.40% on average, and at most 96.38%, respectively, compared to OpenNF.
- *How well does ScaleFlux work with various time window sizes in FIT (Section 5.6)?* We evaluate the FCT with various time window sizes from 20ms to 2000ms. Observing that the average FCT across all the network traces reduces along with the time window size increasing from 20ms to 1000ms. Yet, when the time window size changes from 1000ms to 2000ms, the average FCT changes vary little.
- *How much performance loss does ScaleFlux have when replacing the flow bandwidth predictor or SAA with naive solutions (Section 5.7)?* We compare ScaleFlux with ScaleFlux-FBP and ScaleFlux-SAA. ScaleFlux-FBP and ScaleFlux-SAA are ScaleFlux without flow bandwidth predictor and ScaleFlux without SAA, respectively. The performance loss of ScaleFlux in terms of downtime, buffer usage, and FCT are 63.93% and 2.64%, 77.24% and 28.21%, and 51.52% and 36.15% on average, compared against ScaleFlux-FBP and ScaleFlux-SAA, respectively. Also, ScaleFlux-SAA provides worse per-flow SLAs.
- *How much overhead does ScaleFlux add (Section 5.8)?* We evaluate ScaleFlux's overhead in terms of number of entries used in FIT, model prediction time, and the operating time of flow routing table. The experiment results show that the overhead of FIT is bounded in a reasonable level, and the time used for flow bandwidth prediction and operating flow routing table is limited, which can be ignored compared to flow completion time.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

5.1 Testbed and Setup

Our testbed consists of five servers arranged as shown in Fig. 3a. Each server has two 2.5GHz Intel Haswell 12-Core E5-2680 v3 processors, two Tesla K40m GPUs, 256GB DDR4 RAM, two 1TB 7.2K RPM 3G SATA HDDs, a dual-port Intel X710 10GbE NIC, and a dual-port Mellanox ConnectX-5 10GbE NIC. Two servers run two OpenNF-modified PRADS [30] as VNFs separately. One server runs Open-vSwitch as an OpenFlow switch. Another one runs the ScaleFlux controller while the fifth server generates traffic.

Workloads. We use four industrial workloads including cloud gateway, web search [27], Facebook[28], and the Azure WAN[29] to generate traffic traces. Fig. 3b shows the flow size distributions for the four workloads. For the web search, Facebook, and Azure WAN workloads, since we do not have their concrete flow information, such as bandwidth, flow source and destination IP addresses, and flow arrival times, and only have the flow size distributions, we generate the network traces to satisfy the flow size distributions of the corresponding workloads. Besides, for each industrial workload, we generate 10 network traces with the network loads ranging from 0.1 (i.e., 1Gbps) to 1 (i.e., 10Gbps). For each generated network trace, flows arrive and finish dynamically according to a Poisson process. Specially, for the cloud gateway workload, we have the concrete flow information and it is a 1-hour network trace collected every 1ms from 18:00pm to 19:00pm. The overall network load of the 1-hour network trace is shown in Fig. 1. We scale the flow bandwidth accordingly to match the corresponding network loads.

Methodology. We generate five network traces for each workload above and send flows according to the traces. The same trace is used for the different compared schemes. Thus for each scenario we repeat experiments for five independent runs. We calculate the average result of the five runs for each workload and further obtain the average result for the four workloads, and then report the final average one. We consider VNF queue size thresholds from 10 to 100 packets for the corresponding network loads from 0.1 to 1. We use the corresponding threshold values to calculate the maximum latency that flows can accept to wait for VNF processing.

Schemes Compared. To demonstrate the efficiency of ScaleFlux's scaling scheduling algorithm, we compare it against the optimal one in terms of resource efficiency and flow-level latency. Besides, to demonstrate the ScaleFlux's efficiency in state migration, we compare ScaleFlux against OpenNF that moves all flows. We choose OpenNF because only it provides loss-free and order-preserving guarantees compared with other solutions mentioned in Section 6. These guarantees are important to ensure the new VNF instances work correctly. ScaleFlux also provides these guarantees. In our experiments, we set the guarantee to be loss-free, the scope to be per-flow state, and the optimizations to be parallelizing and early-release for both schemes. This is consistent with the setup in [14]. In addition, we do not compare ScaleFlux against the white-box schemes [11], [12], [22], because they require to totally re-implement VNFs with their own APIs and cannot guarantee VNF processing performance, compared to existing productions.

Metric Used. We utilize detection time to evaluate the efficiency of load monitor surrogate. And we adopt Root Mean Square Error (RMSE) to evaluate the performance of flow bandwidth predictor. RMSE is calculated as the following:

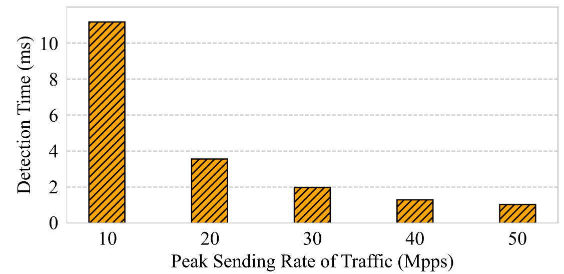


Fig. 10. Detection time of traffic load burstiness for cloud gateway workload with different peak sending rates.

$$RMSE = \left(\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|^2\right)^{\frac{1}{2}}, \quad (28)$$

where y is the actual flow bandwidth time-series, and $\hat{y}$ is the predicted time-series.

5.2 Efficiency of Load Monitor Surrogate

We first investigate the efficiency of ScaleFlux's load monitor surrogate for detecting traffic load burstiness. We proactively generate flows with the same average rate ($\tilde{5}$ Mpps) but different peak rates (e.g., 10, 20, 30, 40, 50 Mpps) regarding to the flow size distributions of the four workloads as shown in Fig. 3b. When the traffic sent from the sender begins to burst, we inject some special signal packets into the traffic. The detection time of burstiness is just the time interval between the time point when the traffic burstiness happens and the one when the scaling scheduler receives the signal packet.

Fig. 10 shows the traffic load detection time for cloud gateway workload with different peak sending rates. In general, other three workloads have the similar variation tendency with different sending peak rates. That is, the detection time decreases as the flow peak rate increases. Table 2 shows the average traffic load detection time across all the four workloads. We can see the average load detection time across various peak sending rates is 3.96ms, which is much smaller than the duration time of traffic burstiness, i.e., several seconds or even minutes.

5.3 Performance of Flow Bandwidth Predictor

In this section, we are going to investigate the performance of ScaleFlux's flow bandwidth predictor. First, we need to determine the proper hyperparameters for ABCNN-LSTM. As we know, for the federated training, proper hyperparameter setup, i.e., threshold of absolute gradient value, is important for the model accuracy and communication efficiency. Therefore, we begin with looking at the performance of ABCNN-LSTM model with different thresholds of gradient and find the most proper one. We select the gradient thresholds from the set $[0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.9, 1, 10]$. Besides, for the model evaluation, we randomly select 70% flows from each network trace and use them for the model training and the remaining for testing.

Fig. 11 shows the average RMSE of all network loads with all the four workloads. We can observe its prediction

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Average Detection Time of Traffic Load Burstiness Across All the Four Workloads With Different Peak Sending Rates

Peak Sending Rate (Mpps)	10	20	30	40	50
Avg. Detection Time (ms)	10.81	4.13	2.27	1.69	0.89

performance becomes better with higher gradient threshold. Besides, when it is 0.25, RMSE is 3.48; and when it is 10, RMSE is 3.439. This means gradient threshold is increased by 40×, the model accuracy is only improved by 1.18%. In addition, we know the gradient threshold is related to communication efficiency and models are usually trained slower with the larger one. Thus, to achieve the best tradeoff between model performance and training efficiency, we choose 0.25 as the final gradient threshold for ABCNN-LSTM.

We now evaluate the performance of ScaleFlux's flow bandwidth predictor, compared against CNN-LSTM [47], LSTM [37], GRU [48], Support Vector Regression (SVR) [49], and ARIMA [50]. We select these models as the baselines of ScaleFlux's flow bandwidth predictor, since they are commonly used in various time-series prediction tasks. All the compared schemes use the same training and testing datasets with ScaleFlux's ABCNN-LSTM. Fig. 12 shows the performance comparison in terms of average RMSE between ABCNN-LSTM, CNN-LSTM, LSTM, GRU, SVR, and ARIMA, for each workload. Observing that the proposed model ABCNN-LSTM performs the best for flow bandwidth time-series prediction. ABCNN-LSTM performs better than CNN-LSTM by 34.04% on average. They are 39.86% for LSTM, 41.52% for GRU, 79.80% for SVR, and 82.85% for ARIMA, respectively. This is because ABCNN-LSTM utilizes attention mechanism to extract important features from the historical bandwidth time-series and avoid the gradient vanishing and exploding problems in deep learning models, such as LSTM and GRU. Besides, ABCNN-LSTM takes advantage of LSTM in predicting time-series data. Especially, ABCNN-LSTM, CNN-LSTM, and LSTM use the completely same LSTM structure here, and LSTM uses the full features without any selection. The performance improvements of both ABCNN-LSTM and CNN-LSTM to LSTM further demonstrate the effectiveness of the ABCNN and CNN-based feature selection methods.

5.4 Efficiency of Scaling Scheduler

We now look at the efficiency of SAA for VNF scaling scheduling. To see the performance gap between SAA and

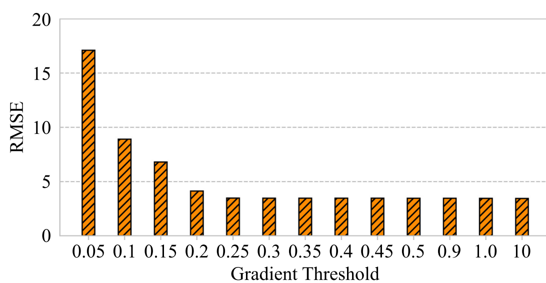


Fig. 11. The average RMSE of ABCNN-LSTM model for flow bandwidth prediction with all the four workloads.

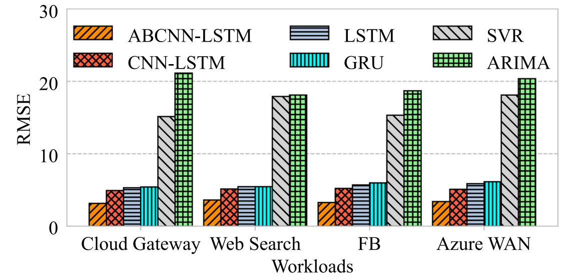


Fig. 12. Performance comparison in average RMSE between ABCNN-LSTM, CNN-LSTM, LSTM, GRU, SVR, and ARIMA, for each workload.

the optimal, we compare the CPU resource usage of SAA against the one of the exiting algorithm for global optimization - the controlled random search (CRS) algorithm [51], which is implemented in the non-linear optimization (NLopt) package in R [52]. Generally, CRS starts with a random "population" of points, and randomly "evolve" these points by heuristic rules. In the experiments, we set the initial population as 3.

Fig. 13 shows the comparison between SAA and the optimal algorithm with various network loads. Here, we show the average CPU usages for each network load across the four workloads. We obtain the total usage results of VNF instances by sending traffic to achieve the specified network load within the first two seconds. When the network load is stable at the target one, we measure the total CPU usages of VNF instances instantiated by SAA and the optimal algorithm, respectively. We can see SAA achieves near-optimal performance in CPU resource usage (The line for SAA and the one for the optimal in Fig. 13 almost coincide). This demonstrates SAA's near-optimal scheduling efficiency in CPU resource usage.

Besides, we then look at flow latency reduction of SAA, during flow and state migration, compared against state-of-the-art black-box scheme OpenNF [14]. We compare ScaleFlux against OpenNF, because it is practical for flow and state migration in NFV and both OpenNF and ScaleFlux do not need to reimplement VNFs from scratch. Other schemes generally adopt white-box methods and require people spending much time learning them. This is impractical, because industry products usually adopt existing open-source projects and different VNFs may use different development frameworks and programming languages. These VNFs, however, belong to the same service chain. Fig. 14 shows the average flow latency reductions with various network loads across the four workloads, compared to OpenNF. We can see SAA achieves at least 217% average

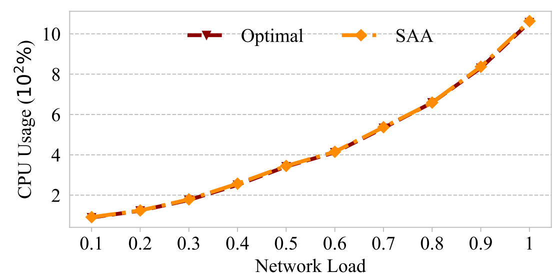


Fig. 13. CPU resource usage comparison between SAA and the optimal algorithm with various network loads. We show the average CPU usage across the four workloads for each network load.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

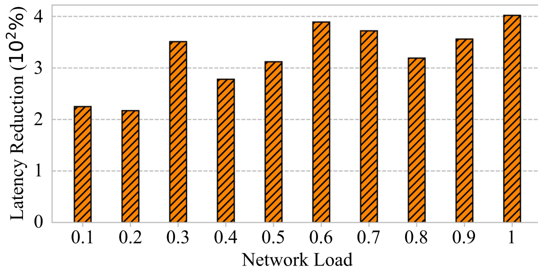


Fig. 14. Average flow latency reductions with various network loads compared against OpenNF. We show the average flow latency reduction for each network load across the four workloads.

latency reduction and at most 402% reduction. This is because that SAA considers flow-level latency constraints when scheduling flows and states to migrate, while minimizing VNF resource usage. Instead, OpenNF simply moves all the flows and states without such consideration. This demonstrates the efficiency of SAA in terms of flow latency reduction.

Finally, we evaluate the efficiency of SAA in terms of execution time over various network loads. Fig. 15 shows the average execution time of SAA to make scheduling decisions with various network loads for the four workloads. Note that the network load represents the total traffic for SAA to schedule and the execution time is the time period elapsed for ScaleFlux to run SAA in order to produce the scheduling results. We can see for various network loads, the execution time of SAA ranges from 2.93ms to 4.37ms, which is significantly smaller than flow completion time and time window (1000ms) used by flow monitor in Section 3.3 to monitor active flows. Also, compared to the duration of traffic load burstiness (i.e., several minutes), the execution time of SAA can be ignored. Therefore, not only SAA can achieve near-optimal performance for CPU resource usage, but also is executed efficiently.

5.5 Downtime, Buffer Usage, and FCT

We now evaluate the performance benefit of ScaleFlux in terms of downtime, buffer usage, and flow completion time (FCT). We use them as the metrics because they can measure and reflect the end-to-end performance of ScaleFlux. Scaling downtime here is defined as time elapsed between the beginning of the `moveFlow` call and the finish time of the last `delPerflow` call. Buffer usage refers to the overall memory used to buffer packets that arrive during flow and state migration. FCT reduction is defined as the ratio between the migrated flow's FCT in OpenNF and the corresponding one in ScaleFlux, and it can evaluate ScaleFlux's end-to-end impact on the performance of migrated flows.

Figs. 16 and 17 show the comparisons in terms of the average downtime and buffer usage, respectively, across various network loads for the four workloads. We can observe that ScaleFlux provides significant performance benefits. It reduces the downtime by at least 60.12%, 72.90% on average, and up to 80% compared to OpenNF, and saves at least 80.50% buffer usage, 83.48% on average, and up to 87.87%. This is because ScaleFlux takes considerations of flow-level latency constraints and selectively migrate flows and states, while significantly reducing the network load at the original VNF instance, instead of moving all like OpenNF. The results demonstrate ScaleFlux can manage VNF scaling with efficient buffer usage.

Then, we look at ScaleFlux's FCT reduction to the migrated flows. We calculate the FCTs for all migrated flows in ScaleFlux and OpenNF, respectively. Note that FCTs in both ScaleFlux and OpenNF include state migration time, as migrated flows' FCTs are affected by the state migration process. Fig. 18 shows the results for average FCT reduction of migrated flows in ScaleFlux across network loads for the four workloads, compared to OpenNF. We can observe that FCTs of migrated flows in OpenNF are extended at least 8.70×, compared to that in ScaleFlux, 17.04× on average, and at most 27.6×. This means ScaleFlux reduces extra end-to-end latency to migrated flows by at least 88.51%, 93.40% on average, and at most 96.38%, respectively. This is because OpenNF moves much more useless flows and states to release the network load at the original VNF instance than ScaleFlux.

5.6 Sensitivity Analysis

As described in Section 3.3, ScaleFlux maintains a flow information table (FIT) to store flows and their bandwidth

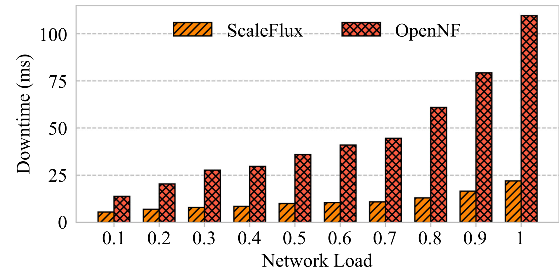


Fig. 16. Average downtime across various network loads for the four workloads.

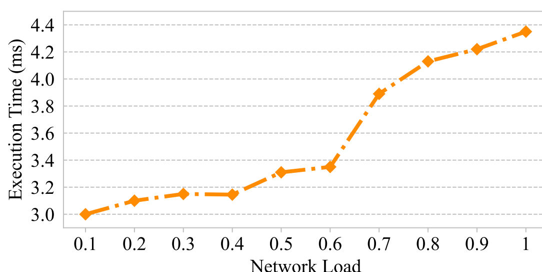


Fig. 15. The execution time of SAA with various network loads. We show its average execution time for finishing one scheduling decision across the four workloads.

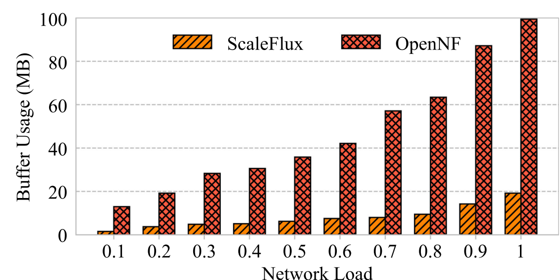


Fig. 17. Average buffer usage across various network loads for the four workloads.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

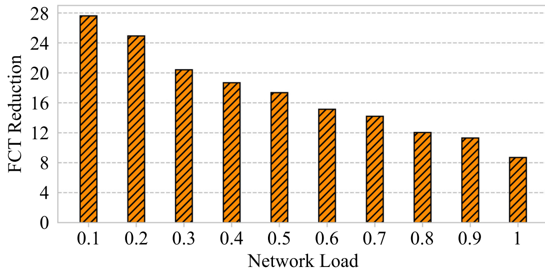


Fig. 18. Average FCT reduction ratio of migrated flows between ScaleFlux and OpenNF across various network loads for the four workloads.

time-series. FIT utilizes a time window to monitor active flows. Therefore, the time window size determines the number of active flows that ScaleFlux can see and further affects the flows ScaleFlux can migrate when VNF scaling is launched. In other words, when the VNF instance is overloaded and scaling is launched, ScaleFlux can only migrate flows from the FIT. Thus, a smaller time window will lead to less migrated flows and launch scaling more repeatedly, because the network load is not migrated crisply for one time. This further leads to that more packets need to buffer at the original VNF instance and results in longer FCTs.

As a result, to find a suitable time window size for all the network traces, we test the FCT with different time window sizes from 20ms to 2000ms. Fig. 19 shows the average FCT with different time window sizes across the four workloads. Note that we increase the time window size with 10ms each time and only show the key points in the figure, and we calculate the average FCT for all the flows here. We can see the average FCT reduces along with the time window size increasing from 20ms to 1000ms. Yet, when the time window size changes from 1000ms to 2000ms, the average FCT changes very little for each workload. These are because ScaleFlux with smaller time window cannot monitor all the active flows, only moves limited traffic for each scaling process, and cannot solve the overload throughly. When the time window size is between 1000ms and 2000ms, ScaleFlux can basically see the same active flows across all the network traces. Thus, ScaleFlux can solve the overload issue for one-shot scaling and achieve stable performance. These demonstrate that 1000ms is the best choice of the time window size for the network traces we use.

5.7 Deep Dive

Next, we conduct a series of experiments to investigate the impact of the key design components in ScaleFlux. We selectively disable each component and use a naive implementation to replace it in ScaleFlux. Here, we compare ScaleFlux against ScaleFlux-FBP and ScaleFlux-SAA. For ScaleFlux-FBP, we disable the flow bandwidth predictor (FBP) in ScaleFlux and use the instant flow bandwidth when scaling is launched as the prediction result. ScaleFlux-SAA uses the top k mechanism to replace SAA in ScaleFlux. For each scaling, ScaleFlux-SAA directly moves top k flows and their states which make the network load lower than the pre-defined threshold.

Figs. 20, 21 and 22 show the comparisons in downtime, buffer usage, and FCT between ScaleFlux, ScaleFlux-FBP, and ScaleFlux-SAA. Observing that compared against ScaleFlux-FBP, ScaleFlux reduces downtime, buffer usage, and FCT by 63.93%, 77.24%, and 51.22% on average, respectively. ScaleFlux's reductions are 2.64%, 28.21%, and 36.15%, compared against ScaleFlux-SAA. Note that we measure the FCTs for all the flows here, instead of only migrated flows.

We can see removing the key component in ScaleFlux (flow bandwidth predictor or SAA) leads to large performance degradation. This is because the non-accurate flow bandwidth prediction can result in that more smaller flows are migrated, which cannot solve the overload issue at once and launches scaling repeatedly in a short time. Further, this causes more flows are migrated than ScaleFlux and thus significantly increases the downtime, buffer usage, and average FCT. ScaleFlux-SAA works better than ScaleFlux-FBP in downtime, buffer usage, and FCT, and has similar performance in downtime with ScaleFlux. Because ScaleFlux-SAA can predict flow bandwidth accurately and the top k mechanism in ScaleFlux-SAA migrates less flows than ScaleFlux-FBP. Yet, migrating top k elephant flows makes ScaleFlux-SAA utilize more buffer and incurs large average FCTs, since the top k elephant flows' traffic and FCTs dominate the overall load and the total FCT. In addition, ScaleFlux can provide better SLA for each flow than ScaleFlux-SAA, as the SAA takes per-flow latency into consideration in the scaling scheduling.

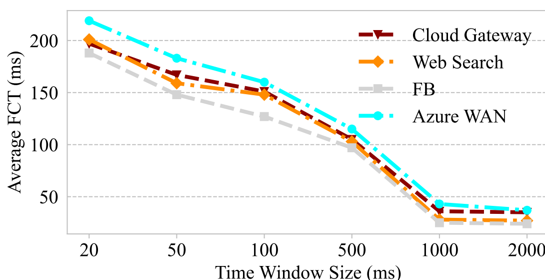


Fig. 19. Average FCT with different time window sizes in FIT across the four workloads. Note that we measure the FCTs for all the flows here, instead of only migrated flows above.

5.8 Overhead

We now evaluate the overhead of ScaleFlux. We first consider the overhead of FIT used in flow monitor. Table 3 shows the number of entries in FIT at each 1000ms-eviction epoch for one run of our experiment using the cloud gateway workload, web search workload, Facebook workload, and the Azure WAN workload, respectively. Results for other runs are qualitatively similar. It indicates that the

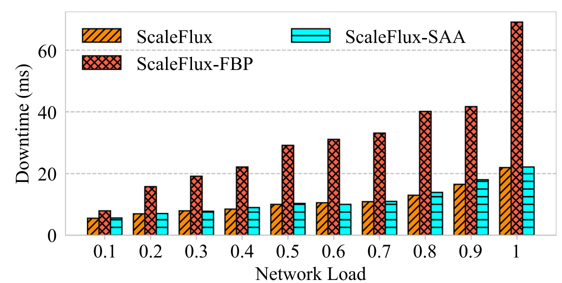


Fig. 20. Average downtime across various network loads for the four workloads.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

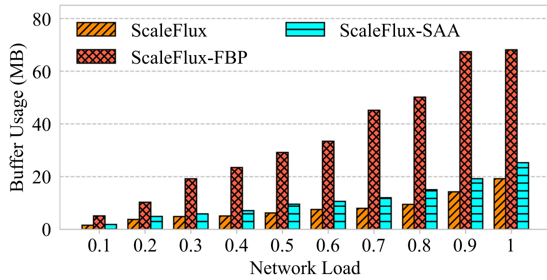


Fig. 21. Average buffer usage across various network loads for the four workloads.

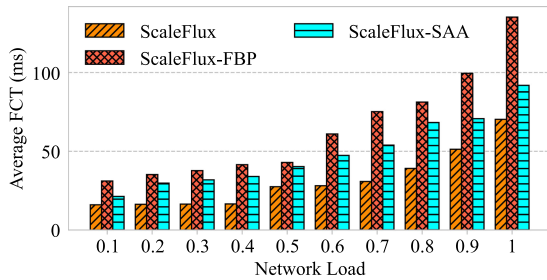


Fig. 22. Average FCT of all the flows across various network loads for the four workloads. Note that we measure the FCTs for all the flows here, instead of only migrated flows.

TABLE 3
The Number of Flow Table Entries in One Run of the Experiment Using the Four Workloads, Respectively

Time (x1000ms)	1	2	3	4	5	6
Number of Entries (Cloud Gateway)	329	601	875	538	402	672
Number of Entries (Web Search)	893	1004	1131	807	891	1029
Number of Entries (Facebook)	503	546	474	553	631	669
Number of Entries (Azure WAN)	424	454	566	403	592	497

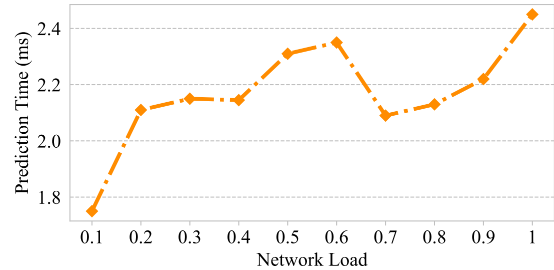


Fig. 23. The average prediction time of ABCNN-LSTM over various network loads for the four workloads.

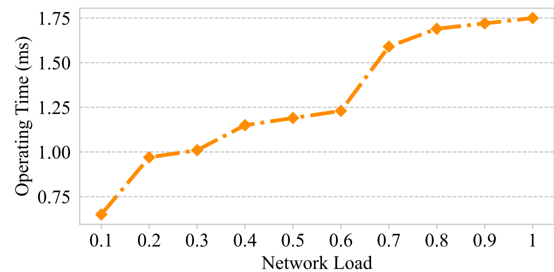


Fig. 24. The average operating time to adjust the flow routing table at OpenVSwitch over various network loads for the four workloads.

overhead of flow information table used by flow monitor is bounded in a reasonable level.

Next, we consider the prediction time of ABCNN-LSTM. As we stated in Section 4, the ABCNN-LSTM model is running on one Tesla K40m GPU for online prediction. Here we only measure the prediction time, i.e., inference time, of the model, and ignore the time used of model training phrase. Because the model training can be done offline, and will not affect the performance of online VNF scaling. Fig. 23 shows the average prediction time of ABCNN-LSTM over various network loads for the four workloads. Besides, in the experiments, the network load is higher, the number of flows to be predicted by the model is more. We can see the prediction time ranges from 1.69ms to 2.45ms, which demonstrates it works efficiently over various network loads.

Finally, we loot at the operating time for the flow routing table at OpenVSwitch to complete flow migration. We obtain the operating time by measuring the total time used to adjust the flow routing table during VNF scaling. Fig. 24 shows the results. We observe the operating time ranges from 0.65ms to 1.75ms over various network loads, which is pretty short. This is because ScaleFlux carefully selects the flows to be migrated and thus significantly reduces the number of table entries to operate.

6 Related Work

Many solutions are proposed to make elastic and flexible VNF scaling in NFV. We present ScaleFlux first in the conference paper [1], which focuses on efficient state migration in NFV, instead of a NFV scaling system. Here we significantly revise the design to make it become a complete end-to-end NFV scaling system to support real-time traffic load burstiness detection and corresponding elastic stateful VNF scaling to achieve both flow-level latency guarantee and VNF resource efficiency (Section 3). We implement ScaleFlux (Section 4) and evaluate it using testbed experiments (Section 5). Of course, there are some recent work which focuses on different aspects of VNF scaling. We discuss them in the following.

Elastic NFV Scaling. E2 [10], NFVnice [53], Metron [12], ScalIMS [54] and FlexNFV [9] consider dynamic scaling to improve service chain performance. To avoid complex state migration, E2 [10] only uses new created VNFs to process new flows without states, while FlexNFV [9] adopts consistent hashing to keep the flow affinity. NFVnice [53] utilizes backpressure mechanism and proactively adjusts CPU allocation to the service chain to improve chain-level performance and avoid resource wastage. Flurries [55] runs a unique service chain for each flow to provide flow-level customized SLA. ScaleFlux can easily support VNF service chains like these work without the need to do any modifications, as the service chain can be seen as a consolidated VNF [56], [57]. Besides, ScaleFlux can automatically detect network load changes and trigger stateful VNF scaling in real time. For scenarios where existing flows' SLAs cannot be guaranteed by the existing VNFs, ScaleFlux can adaptively perform vertical scaling. Besides, some works are proposed to minimize VNF deployment and placement cost [58], [59], [60], [61], [62], [63], [64], [65]. ScaleFlux tries to minimize the CPU resource usage for VNF scaling.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

Similar to [9], [10], [53], [66], [67], ScaleFlux makes automatic VNF scaling with network load changing. Yet, ScaleFlux achieves efficient stateful scaling and tries to reduce the latency of flow migration.

NFV State Migration. There are many existing solutions for NFV state management [68], [69]. Split/Merge [13], Pico Replication [70], OpenNF [14], [71], and DiST [72] are systems that provide some control over both internal VNF state and network state. Split/Merge and Pico Replication provide shared libraries that VNFs use to create, access, and modify internal state through pre-defined APIs. OpenNF provides a northbound API for applications to specify which state to move, and which guarantees to enforce; and it also implements a southbound API for the controller to perform the export or import of VNF state. DiST [72] differs from OpenNF by buffering the packets during migration at the destination VNF instead of at the controller. Olteanu and Raiciu [73] attempt to migrate per-flow state between VM replicas without application modifications. Similar to [13], [14], [71], ScaleFlux achieves efficient NFV state migration. Further, ScaleFlux proposes a complete VNF auto-scaling framework to enable efficient VNF scaling including state management.

White-Box NFV. Some research work[11], [12], [22], [74] proposes and builds a complete white-box NFV framework and APIs to manage VNFs. For example, StatelessNF [22] re-architects network functions so that their internal states are maintained in a shared separate storage tier, which has to face the performance challenge for frequent state update. S6 [11] proposes a distributed shared state model and decouples state maintainance and VNF processing. Metron [12] splits traffic into different classes and takes traffic class as the basic unit to perform packet processing, I/O operations, and state management. Yet, the white-box NFV frameworks require to completely re-implement VNFs, which is hard in practice, requires long learning process and huge development effort. In contrast, ScaleFlux provides a black-box model without the need to re-implement VNFs and separates the implementation of VNF products and scaling management, and thus frees VNF operators to focus on the packet processing logic.

In sum, ScaleFlux designs and implements a complete system to enable end-to-end automatic stateful VNF scaling, while achieving resource efficiency and flow-level latency guarantee.

7 Conclusion

We introduced ScaleFlux, an efficient end-to-end NFV system to achieve automatic VNF scaling. ScaleFlux achieves resource efficiency and flow-level latency guarantee by adopting queue-based network load detection mechanism to launch VNF scaling timely, designing an ABCNN-LSTM model for flow bandwidth prediction, and leveraging the simulated annealing algorithm for efficient scaling scheduling. ScaleFlux efficiently migrates per-flow state for selected flows and maintains state for other flows in the original VNF instance until they expire. We implemented ScaleFlux and evaluated it on an five-machine testbed. Our experiment results show that ScaleFlux significantly achieves efficient resource usage for VNFs and reduces downtime and the performance penalty during scaling.

References

[1] L. Liu, H. Xu, Z. Niu, P. Wang, and D. Han, “U-HAUL: Efficient state migration in NFV,” in *Proc. 7th ACM SIGOPS Asia-Pacific Workshop Syst.*, 2016, pp. 1–8.

[2] European Telecommunications Standards Institute, Network functions virtualisation: Introductory white paper, 2012. [Online]. Available: http://portal.etsi.org/NFV/NFV_White_Paper.pdf

[3] X. Fei, F. Liu, H. Jin, and H. Hu, “Paving the way for NFV acceleration: A taxonomy, survey and future directions,” *ACM Comput. Surveys*, vol. 53, no. 4, pp. 1–42, 2020.

[4] S. Jain et al., “B4: Experience with A globally-deployed software defined WAN,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, 2013.

[5] U. Naseer, L. Niccolini, U. Pant, A. Frindell, R. Dasineni, and T. A. Benson, “Zero downtime release: Disruption-free load balancing of a multi-billion user website,” in *Proc. ACM Annu. Conf. ACM Special Int. Group Data Commun. Appl., Technol., Architect., Protoc. Comput. Commun.*, 2020, pp. 529–541.

[6] H. Shao, X. Wang, Y. Lu, Y. Yu, S. Zheng, and Y. Zhao, “Accessing cloud with disaggregated software-defined router,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 1–14.

[7] S. Yan, X. Wang, X. Zheng, Y. Xia, D. Liu, and W. Deng, “ACC: Automatic ECN tuning for high-speed datacenter networks,” in *Proc. ACM SIGCOMM Conf.*, 2021, pp. 384–397.

[8] Z. Ye, Y. Wang, S. He, C. Xu, and X.-H. Sun, “Sova: A software-defined autonomic framework for virtual network allocations,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 116–130, Jan. 2021.

[9] X. Fei, F. Liu, H. Jin, and B. Li, “FlexNFV: Flexible network service chaining with dynamic scaling,” *IEEE Netw.*, vol. 34, no. 4, pp. 203–209, Jul./Aug. 2020.

[10] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, and S. Ratnasamy, “E2: A framework for NFV applications,” in *Proc. ACM 25th Symp. Oper. Syst. Princ.*, 2015, pp. 121–136.

[11] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, “Elastic scaling of stateful network functions,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 299–312.

[12] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr, “Metron: NFV service chains at the true speed of the underlying hardware,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 171–186.

[13] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, “Split/Merge: System support for elastic execution in virtual middleboxes,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2013, pp. 227–240.

[14] A. Gember-Jacobson et al., “OpenNF: Enabling Innovation in Network Function Control,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 163–174, 2014.

[15] W. Wang, X. C. Wu, P. Tammana, A. Chen, and T. S. E. Ng, “Closed-loop network performance monitoring and diagnosis with spidermon,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 267–285.

[16] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo, “BurstRadar: Practical real-time microburst monitoring for datacenter networks,” in *Proc. ACM Asia-Pacific Workshop Syst.*, 2018, pp. 1–8.

[17] V. Nathan, V. Sivaraman, R. Addanki, M. Khani, P. Goyal, and M. Alizadeh, “End-to-end transport for video QoE fairness,” in *Proc. ACM Special Int. Group Data Commun.*, 2019, pp. 408–423.

[18] X. Chen, H. Kim, J. M. Aman, W. Chang, M. Lee, and J. Rexford, “Measuring TCP round-trip time in the data plane,” in *Proc. ACM Workshop Secure Programmable Netw. Infrastructure*, 2020, pp. 35–41.

[19] X. Zuo et al., “Bandwidth-efficient multi-video prefetching for short video streaming,” 2022, *arXiv:2206.09839*.

[20] W. Yin, H. Schütze, B. Xiang, and B. Zhou, “ABCNN: Attention-based convolutional neural network for modeling sentence pairs,” *Trans. Assoc. Comput. Linguistics*, vol. 4, pp. 259–272, 2016.

[21] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller, “Stateless network functions,” in *Proc. ACM SIGCOMM Workshop Hot Topics Middleboxes Netw. Function Virtualization*, 2015, pp. 49–54.

[22] M. Kablan, A. Alsudais, E. Keller, and F. Le, “Stateless network functions: Breaking the tight coupling of state and processing,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 97–112.

[23] J. Sherry and S. Ratnasamy, “A survey of enterprise middlebox deployments.” 2012. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012–24.pdf

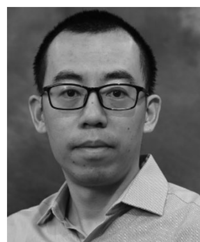
[24] S. Hanks, D. Meyer, D. Farinacci, T. Li, and P. Traina, “Generic routing encapsulation (GRE),” *RFC 2784*, 2000. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2784.html

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

[25] M. Mahalingam et al., "Virtual eXtensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks," *RFC 7348*, pp. 1–22, 2014.
[26] D. Kim et al., "TEA: Enabling state-intensive network functions on programmable switches," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2020, pp. 90–106.
[27] M. Alizadeh et al., "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 63–74.
[28] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2015, pp. 123–137.
[29] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart, "Next stop, the cloud: Understanding modern web service deployment in EC2 and Azure," in *Proc. Conf. Internet Meas. Conf.*, 2013, pp. 177–190.
[30] Passive real-time asset detection system. 2009. [Online]. Available: http://prads.projects.linpro.no
[31] Intel, "Data plane development kit," 2014. [Online]. Available: https://www.intel.com/content/www/us/en/developer/topic-technology/networking/dpdk.html
[32] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, Art. no. 9.
[33] 802.1Qbb, "802.1Qbb - Priority-based flow control," 2008. [Online]. Available: http://www.ieee802.org/1/pages/802.1bb.html
[34] C. Tian et al., "P-PFC: Reducing tail latency with predictive PFC in lossless data center networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 6, pp. 1447–1459, Jun. 2020.
[35] Y. LeCun et al., "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, vol. 1, no. 4, pp. 541–551, 1989.
[36] DPDK. 2013. [Online]. Available: http://dpdk.org/
[37] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[38] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea, "Performance contracts for software network functions," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2019, pp. 517–530.
[39] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Archit. Protoc. Comput. Commun.*, 2020, pp. 270–282.
[40] J. Gong, Y. Li, B. Anwer, A. Shaikh, and M. Yu, "Microscope: Queue-based performance diagnosis for network functions," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Archit. Protoc. Comput. Commun.*, 2020, pp. 390–403.
[41] K. Jin and C. Xia, "Data-driven design of microtransit services via optimal transport and simulated annealing," in *Proc. 22nd Int. Workshop Mobile Comput. Syst. Appl.*, 2021, pp. 179–181.
[42] A. W. Harrow and A. Y. Wei, "Adaptive quantum simulated annealing for Bayesian inference and estimating partition functions," in *Proc. 31st Annu. ACM-SIAM Symp. Discrete Algorithms*, 2020, pp. 193–212.
[43] C. Song et al., "HDFI: Hardware-assisted data-flow isolation," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 1–17.
[44] A. Wang, Y. Guo, F. Hao, T. V. Lakshman, and S. Chen, "UMON: Flexible and fine grained traffic monitoring in open vSwitch," in *Proc. ACM Conf. Emerg. Netw. Experiments Technol.*, 2015, Art. no. 15.
[45] S. Zhao et al., "I-CaN-MaMa: Integrated campus network monitoring and management," in *Proc. IEEE Netw. Operations Manage. Symp.*, 2014, pp. 1–7.
[46] G. Liu, S. Guo, B. Xiao, and Y. Yang, "SDN-based traffic matrix estimation in data center networks through large size flow identification," *IEEE Trans. Cloud Comput.*, vol. 10, no. 1, pp. 675–690, First Quarter 2022.
[47] J. Donahue et al., "Long-term recurrent convolutional networks for visual recognition and description," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 2625–2634.
[48] K. Cho et al., "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1724–1734.
[49] H. Drucker et al., "Support vector regression machines," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 1997, pp. 156–161.
[50] A. J. Conejo, M. A. Plazas, R. Espinola, and A. B. Molina, "Day-ahead electricity price forecasting using the wavelet transform and ARIMA models," *IEEE Trans. Power Syst.*, vol. 20, no. 2, pp. 1035–1042, May 2005.
[51] W. Price, "Global optimization by controlled random search," *J. Optim. Theory Appl.*, vol. 40, no. 3, pp. 333–348, 1983.
[52] NLopt algorithms. 2018. [Online]. Available: https://nlopt.readthedocs.io/en/latest/NLopt_Algorithms/
[53] S. G. Kulkarni et al., "NFVnice: Dynamic backpressure and scheduling for NFV service chains," *IEEE/ACM Trans. Netw.*, vol. 28, no. 2, pp. 639–652, Apr. 2020.
[54] J. Duan, C. Wu, F. Le, A. X. Liu, and Y. Peng, "Dynamic scaling of virtualized, distributed service chains: A case study of IMS," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11, pp. 2501–2511, Nov. 2017.
[55] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained NFs for flexible per-flow customization," in *Proc. 12th Int. Conf. Emerg. Netw. Experiments Technol.*, 2016, pp. 3–17.
[56] J. Zheng et al., "Orchestrating service chain deployment with Plutus in next generation cellular core," in *Proc. IEEE/ACM 27th Int. Symp. Qual. Serv.*, 2019, pp. 1–10.
[57] C. Tian, A. Munir, A. X. Liu, J. Yang, and Y. Zhao, "OpenFunction: An extensible data plane abstraction protocol for platform-independent software-defined middleboxes," *IEEE/ACM Trans. Netw.*, vol. 26, no. 3, pp. 1488–1501, Jun. 2018.
[58] Z. Li and Y. Yang, "Placement of virtual network functions in hybrid data center networks," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 4, pp. 861–873, Oct.–Dec. 2018.
[59] J. Zheng et al., "Optimizing NFV chain deployment in software-defined cellular core," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 2, pp. 248–262, Feb. 2020.
[60] J. Zheng, Z. Zhang, Q. Ma, X. Gao, C. Tian, and G. Chen, "Multi-resource VNF deployment in a heterogeneous cloud," *IEEE Trans. Comput.*, vol. 71, no. 1, pp. 81–91, Jan. 2022.
[61] Y. Xiao et al., "NFVdeep: Adaptive online service function chain deployment with deep reinforcement learning," in *Proc. IEEE/ACM 27th Int. Symp. Qual. Serv.*, 2019, pp. 1–10.
[62] S. Rampersaud and D. Grosu, "Sharing-aware online virtual machine packing in heterogeneous resource clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 2046–2059, Jul. 2017.
[63] X. Shang, Y. Huang, Z. Liu, and Y. Yang, "Reducing the service function chain backup cost over the edge and cloud by a self-adapting scheme," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 2096–2105.
[64] X. Shang, Z. Liu, and Y. Yang, "Online service function chain placement for cost-effectiveness and network congestion control," *IEEE Trans. Comput.*, vol. 71, no. 1, pp. 27–39, Jan. 2022.
[65] X. Shang, Y. Liu, Y. Mao, Z. Liu, and Y. Yang, "Greening reliability of virtual network functions via online optimization," in *Proc. IEEE/ACM 28th Int. Symp. Qual. Serv.*, 2020, pp. 1–10.
[66] R. Han et al., "Workload-adaptive configuration tuning for hierarchical cloud schedulers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 12, pp. 2879–2895, Dec. 2019.
[67] G. Hu, Q. Li, S. Ai, T. Chen, J. Duan, and Y. Wu, "A proactive auto-scaling scheme with latency guarantees for multi-tenant NFV cloud," *Comput. Netw.*, vol. 181, no. 2, 2020, Art. no. 107552.
[68] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella, "Liveness verification of stateful network functions," in *Proc. 17th USENIX Conf. Netw. Syst. Des. Implementation*, 2020, pp. 257–272.
[69] J. Khalid and A. Akella, "Correctness and performance for stateful chained network functions," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 501–515.
[70] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 1.
[71] A. Gember-Jacobson and A. Akella, "Improving the safety, scalability, and efficiency of network function state transfers," in *Proc. ACM SIGCOMM Workshop Hot Topics Middleboxes Netw. Function Virtualization*, 2015, pp. 43–48.
[72] B. Kothandaraman, M. Du, and P. Sköldström, "Centrally controlled distributed VNF state management," in *Proc. ACM SIGCOMM Workshop Hot Topics Middleboxes Netw. Function Virtualization*, 2015, pp. 37–42.
[73] V. A. Olteanu and C. Raiciu, "Efficiently migrating stateful middleboxes," in *Proc. ACM SIGCOMM Conf.*, 2012, pp. 93–94.
[74] J. Duan, X. Yi, J. Wang, C. Wu, and F. Le, "NetStar: A future/promise framework for asynchronous network functions," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 600–612, Mar. 2019.

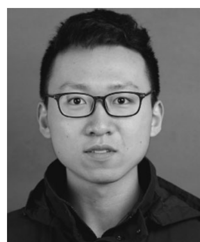
Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.

Libin Liu (Member, IEEE) received the BE degree in software engineering from Shandong University, and the PhD degree from the Department of Computer Science, City University of Hong Kong. He is currently an assistant researcher with Zhongguancun Laboratory, Beijing, China. His current research interests include NFV, resource scheduling for data analytics systems, and machine learning for networking. From 2020 to 2022, he was with Department of Networking Platform in Tencent, Huawei Hong Kong Research Center, and Shandong Computer Science Center (National Supercomputing Center in Ji'nan). He is a member of the ACM.



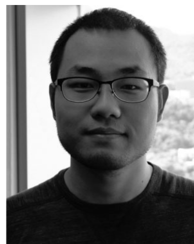
Hong Xu (Senior Member, IEEE) received the BEng degree from the Chinese University of Hong Kong, in 2007, and the MASc and PhD degrees from the University of Toronto, in 2009 and 2013, respectively. He is an Associate Professor with the Department of Computer Science and Engineering, Chinese University of Hong Kong. His research area is computer networking and systems, particularly Big Data systems and data center networks. From 2013 to 2020, he was with the City University of Hong Kong. He was the recipient of an Early Career Scheme Grant from the Hong Kong Research Grants Council in 2014. He received three best paper awards, including the IEEE ICNP 2015 Best Paper Award. He is a senior member of the ACM.

Zhixiong Niu received the BE degree in network engineering from Dalian Maritime University (DMU), in 2012, the MSc degree in computer science from the University of Hong Kong (HKU), in 2014, and the PhD degree from the Department of Computer Science, City University of Hong Kong, in 2019. He is a senior researcher with Microsoft Research Asia.

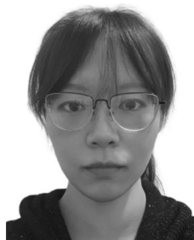


Jingzong Li (Student Member, IEEE) received the BE degree majored in software engineering from the University of Electronic Science and Technology of China, in 2019. He is currently working toward the PhD degree with the Department of Computer Science, City University of Hong Kong. His research interests include video systems and networked systems. He is a student member of the ACM.

Wei Zhang received the BE degree from Zhejiang University, in 2004, the MS degree from Liaoning University, in 2008, and the PhD degree from the Shandong University of Science and Technology, in 2018. He is currently a Professor with the Shandong Computer Science Center (National Supercomputer Center in Ji'nan), Qilu University of Technology (Shandong Academy of Sciences). His research interests include future generation network architectures, edge computing, and edge intelligence.



Peng Wang received the BS degree in information engineering from Xidian University, Xian, China, and the PhD degree from the Department of Computer Science, City University of Hong Kong. He is currently a researcher with Theory Lab, Huawei Hong Kong Research Center, Hong Kong. His research interests include data center networking and cloud computing. He received the Best Paper Award from ACM CoNEXT Student Workshop 2014.



Jiamin Li (Student Member, IEEE) received the BS degree from the Department of Computer Science, City University of Hong Kong, in 2019. She is currently working toward the PhD degree in the Department of Computer Science, City University of Hong Kong. Her research interests include distributed machine learning, machine learning systems, and resource scheduling. She is a student member of the ACM.

Jason Chun Xue (Member, IEEE) received the BS degree in computer science and engineering from the University of Texas at Arlington, in 1997, and the MS and PhD degrees in computer science from the University of Texas at Dallas, in 2002 and 2007, respectively. He is currently an Associate Professor with the Department of Computer Science, City University of Hong Kong. His research interests include memory and parallelism optimization for embedded systems, software/hardware codesign, real time systems, and computer security. He is a distinguished member of the ACM.

Cong Wang (Fellow, IEEE) is currently a Professor with the Department of Computer Science, City University of Hong Kong. His current research interests include data and network security, blockchain and decentralized applications, and privacy-enhancing technologies. He is one of the founding members of the Young Academy of Sciences of Hong Kong. He received the Outstanding Researcher Award (junior faculty) in 2019, the Outstanding Supervisor Award in 2017 and the president's awards in 2019 and 2016, all from the City University of Hong Kong. He is a co-recipient of the IEEE INFOCOM Test of Time Paper Award 2020, Best Paper Award of IEEE ICDCS 2020, Best Student Paper Award of IEEE ICDCS 2017, and the Best Paper Award of IEEE ICPADS 2018 and MSN 2015. His research has been supported by multiple government research fund agencies, including the National Natural Science Foundation of China, Hong Kong Research Grants Council, and Hong Kong Innovation and Technology Commission. He served as associate editor of the *IEEE Transactions on Dependable and Secure Computing*, *IEEE Internet of Things Journal* and *IEEE Networking Letters*, and *The Journal of Blockchain Research*, and TPC co-chairs for a number of IEEE conferences and workshops. He is a member of the ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on December 19,2022 at 06:35:40 UTC from IEEE Xplore. Restrictions apply.