Control Channel Isolation in SDN Virtualization: A Machine Learning Approach

Yeonho Yoo, Gyeongsik Yang, Changyong Shin, Jeunghwan Lee, Chuck Yoo
Department of Computer Science and Engineering, Korea University
{yhyoo, ksyang, cyshin, jhlee21, chuckyoo}@os.korea.ac.kr

***Abstract*—Performance isolation is an essential property that network virtualization must provide for clouds. This study addresses the performance isolation of the control plane in virtualized software-defined networking (SDN), which we call control channel isolation. First, we report that the control channel isolation is seriously broken in the existing network hypervisor in that the end-to-end control latency grows by up to 15× as the number of virtual switches increases. This jeopardizes the key network operations, such as routing, in datacenters. To address this issue, we take a machine learning approach that learns from the past control traffic as time-series data. We propose a new network hypervisor, *Meteor*, that designs an LSTM autoencoder to predict the control traffic per virtual switch. Our evaluation results show that *Meteor* improves the processing latency per control message by up to 12.7×. Furthermore, *Meteor* reduces the end-to-end control latency by up to 73.7%, which makes it comparable to the non-virtualized SDN.**

***Index Terms*—SDN, Network virtualization, Control channel, Isolation, Machine learning, LSTM autoencoder**

I. Introduction

Network virtualization (NV) is essential for delivering isolated network services to users (tenants) in clouds [1]–[4]. In particular, NV based on software-defined networking, called SDN-NV, becomes of paramount importance, as it enables tenants to create their own virtual networks (VNs) on a physical network. The demand for SDN-NV is continuously increasing because the customized VN topology by tenants realizes the network slicing and in-network computing of upcoming network systems [5]–[9].

In SDN-NV, a tenant uses an SDN controller to manage the virtual switches (vSwitches) in its VN. A control channel is logically established between the SDN controller and each vSwitch. Through the control channel, the tenant conducts network controls in vSwitches such as packet forwarding rule setup [10], network monitoring [11], traffic load balancing [12], and custom in-network operations (e.g., boosting the deep learning training [13], [14]). Each network control is achieved through multiple control messages.[1] An SDN controller sends the control messages to vSwitches via the network hypervisor (NH), and the NH translates each control message from the VN context to the physical network context (message translation). Then, the messages are delivered to the physical network to realize the network control. Thus, NH is a key component of SDN-NV.

[1] An individual packet of control traffic is referred to as a "control message." We use the two terms, control traffic and control message, interchangeably.

Recent research has advanced various aspects of NH—VN abstractions [15], [16], scalability [17], and visibility [11], but one remaining issue is "performance isolation." Each tenant requires isolation for its VN to obtain the desired performance in the presence of the other VNs [18], [19]. This is an essential property that NV must provide [20], [21]. As the SDN-NV consists of two layers (i.e., the data and control planes), the isolation is twofold: 1) bandwidth isolation for the data plane and 2) control channel isolation for the control plane. Bandwidth isolation prevents the negative effects on traffic performance (e.g., TCP traffic throughput) caused by the traffic of other tenants. Control channel isolation is to preserve the control traffic performance of each vSwitch, such as message translation latency, even when the control channels increase. Bandwidth isolation is addressed in [19], but to our knowledge, control channel isolation has not been investigated (§VI).

Control channel isolation is a critical challenge in SDN-NV because SDN-NV hosts multiple tenants, and the number of vSwitches (so the number of control channels) inevitably increases. To quantify the control channel isolation, we evaluate an up-to-date open-source NH [5] by varying the number of vSwitches from 8 to 128. This increase of vSwitches is similar to the increase in the number of tenants with respect to the control channel isolation (e.g., when each tenant has 8 vSwitches, the increase in vSwitches from 8 to 128 corresponds to tenants from 1 to 16). We measure the latency for each control message, called the message translation latency (MTL). The MTL is measured as the time elapsed from when a control message arrives in the NH to when the message is sent to the physical network, so it covers the control message processing within the NH. In our experiments, when a tenant performs packet forwarding for a new network connection (forwarding setup), the MTL increases by up to 11.7× as the number of vSwitches increases. These results demonstrate that the control channel isolation is broken seriously.

When control channel isolation is broken, many critical problems arise. One problem is that the MTL contributes to a significant increase in the end-to-end latency of network control (NCL). NCL is the entire time to achieve the network control, which includes the processing of multiple control messages (i.e., the elapsed time from the first control message generated by an SDN controller to the last control message delivered to the physical network). In our experiments, the NCL for forwarding setup increases by up to 12× as the number of vSwitches increases from 8 to 128. Such a signifi-

cant NCL hinders service quality greatly in clouds [22]–[24]. As for network monitoring, we find that NCL increases by up to 15×, which causes critical network management tasks (such as routing, firewall, and traffic load-balancing) to work incorrectly [25]. The results show that the lack of control channel isolation hampers SDN-NV's practicability.

Several studies exist for reducing NCL. In non-virtualized SDNs, existing studies have focused on the physical network or SDN controller side, which can be categorized into two approaches: 1) compressing or aggregating the control traffic [10], [26]–[29] and 2) parallel processing of the control traffic using multi-threads [30]. These approaches are used to reduce the control traffic amount for physical switches or improve controller performance. However, they do not address the control traffic interference between tenants, which breaks the control channel isolation of NH. Therefore, they cannot solve the control channel isolation problem in SDN-NV.

This study presents *Meteor*, a new NH that offers control channel isolation. The central idea of *Meteor* is to calculate and enforce a message translation quota, γ, for a vSwitch. The γ is the amount of control traffic per vSwitch that NH can translate per second. *Meteor* calculates a distinct γ per vSwitch. The most straightforward method for calculating γ is fair share, which is to evenly divide the maximum amount of control traffic that NH can translate per second. However, we find that fair share is very problematic because each vSwitch receives a highly varying amount of control traffic depending on its location in the VN topology or the amount of data plane traffic that the switch handles. Our experiment shows that control traffic varies depending on vSwitches up to 5.3× (§II-C2). As the control traffic greatly differs between vSwitches, we find that fair share is not a proper choice.

Our approach in *Meteor* is to learn the behavior of the control traffic and predict the future control traffic that each vSwitch would receive. We observe that control traffic has a strong relationship between past and future data. So, we treat the control traffic as time-series data, a sequence of control traffic amounts received per second. *Meteor* designs a machine learning (ML) model from this data for prediction. Because there are quite a few ML models, in order to find a suitable one, we carefully evaluate five candidate models and eventually select the LSTM autoencoder (§III-A).

Meteor takes the following inputs: 1) past control traffic to a vSwitch, 2) control message types, and 3) the data plane information for the vSwitch. The LSTM autoencoder is trained with the inputs. The trained model enables *Meteor* to predict the future control traffic of the vSwitch and so to calculate γ. *Meteor* enforces each control channel with γ, which isolates the control channels (§III-C). We implement *Meteor* based on PyTorch along with an open-source NH [5]. *Meteor* achieves the following contributions:

- MTL improves by up to 12.7×.
- NCLs for the forwarding setup and network monitoring improve by up to 3.5× and 3.8×, respectively, which is comparable to those of non-virtualized SDN.

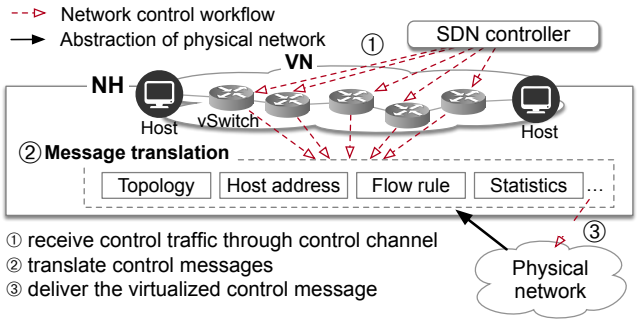


Fig. 1: Network control workflow in SDN-NV. Details in §II-A

- *Meteor* exhibits reasonable overheads—0.4% and 12% increase in memory and CPU cycles, respectively.
- Our design of LSTM autoencoder outperforms other ML models for prediction accuracy by up to 2.2× and is also applicable to complex network topologies.

II. Background and Motivation

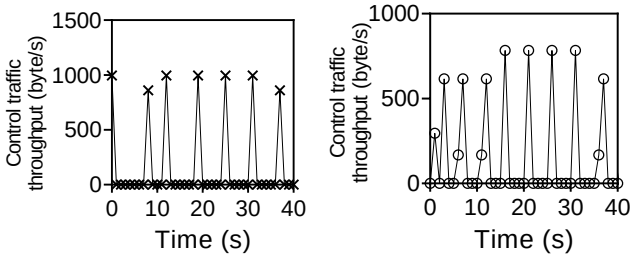
A. SDN-NV

Fig. 1 shows the network control workflow in SDN-NV that sends control traffic (control messages) to a vSwitch. The NH abstracts the physical network (e.g., physical switches and links) and provides a custom VN for each tenant. The SDN controller of a tenant then conducts network controls over the VN. For example, forwarding setup is done with control messages installing a flow rule for every vSwitch that forwards packets. Control messages arrive at the NH (① in Fig. 1). Then, the NH performs the appropriate translations according to the control message type (②). For example, when the type is new flow rule installation, the NH performs topology translation [16], [20] that converts the addresses of vSwitches into the addresses of the physical switches [20]. In addition to topology translation, there are various translations (e.g., host address [16], flow rule [15], and statistics [11]). Note that these message translations are essential to enable network controls of tenants in SDN-NV [5], [15], [16], [20]. Afterward, the control message is delivered to the proper physical switch (③).

So, each control message goes through three steps: 1) transmission of the message from the SDN controller to the NH, 2) message translation in the NH, and 3) transmission of the control message from the NH to the physical switch. We denote the latency for each step as l_V, MTL, and l_P where MTL is the latency overhead generated purely by the NH in SDN-NV. We measure these latencies later (§II-D1).

B. Experiment Setup

We explain the experiment setup used in this study. Our testbed consists of three physical machines equipped with Intel Xeon E5-2600 CPUs and 64 GB memory. The three servers are connected by a 10 Gb Ethernet. We run physical network, NH, and SDN controllers on each of the three servers. The physical network is emulated by Open vSwitch and Mininet [31], and it is of a linear topology with 128 physical switches. For the NH, we run Libera [5], a recent open-source NH, to



(a) Forwarding setup. (b) Network monitoring.

Fig. 2: Control traffic throughput of a vSwitch (bytes/s). Details in §II-C1

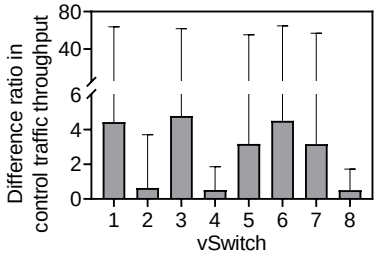


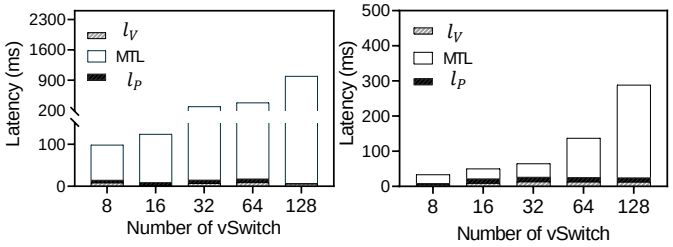
Fig. 3: Difference ratio in control traffic throughput. Details in §II-C2

evaluate the existing SDN-NV. The NH creates a VN with a linear topology, and the number of vSwitches in the VN varies in the range of 8, 16, 32, 64, and 128. Also, the number of hosts in the VN is set to two.

For the SDN controller, we use ONOS, which is widely used in real-world scenarios (e.g., AT&T and COMCAST [32]). The SDN controller performs various network controls, such as topology discovery, forwarding setup, and network monitoring. Topology discovery and network monitoring occur periodically, while forwarding setup occurs when new connections (e.g., TCP) are generated. We focus on the two network controls: 1) forwarding setup and 2) network monitoring because these two network controls are representative metrics in evaluating control channel performance in SDN [10], [11], [25]. Also, these are the network controls that most SDN controllers (e.g., ONOS, OpenDaylight, Floodlight, POX, and Ryu) provide by default. For the testbed, we create a single TCP connection between the two hosts. We employ this experiment setup throughout this paper. We also perform experiments on different topology, the higher number of tenants, and different SDN controller as necessary. The specific differences in experiments are individually described.

C. Control Traffic Characteristics

1) Burst control traffic: To investigate the control traffic characteristics, we experiment with a VN of 8 vSwitches and measure the control traffic throughput of each vSwitch. Fig. 2 shows the control traffic of forwarding setup and network monitoring for a vSwitch over time. It presents the snapshot of the first 40 s of the 120 s measurements because the remaining 80 s shows similar results. Fig. 2a is for the forwarding setup, and we observe that the control traffic throughput shows burstiness ($\times$ marks). Fig. 2b also exhibits burstiness for network monitoring ($\circ$ marks). We experiment with other network controls and observe similar patterns. In other words, the SDN controller generates control traffic in bursts. This burstiness is observed in all vSwitches. Then, when the burstly received messages are translated in the NH, NCL gets increased highly which breaks control channel isolation.



(a) Forwarding setup. (b) Network monitoring.

Fig. 4: Total latency per control message and MTL. Details in §II-D1.

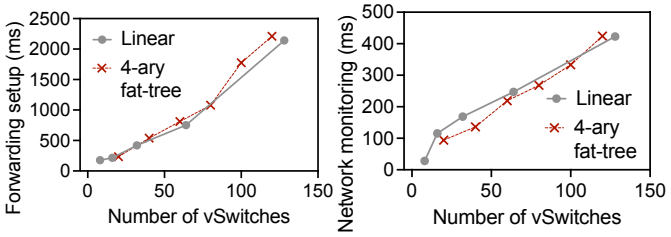
2) Different amount of control traffic per vSwitch: Next, we check the control traffic generated for each vSwitch. We show the control traffic throughput difference between vSwitches by calculating the difference ratio. Fig. 3 shows the difference ratio per vSwitch. The difference ratio 1 means that the vSwitch has received the minimum control traffic between the vSwitches at a second. Also, the difference ratio 4 indicates that the vSwitch has received 4$\times$ higher control traffic than the minimum amount that the other vSwitches have received at a second. The x-axis represents the individual vSwitches (i.e., 1 to 8). The bars are the mean values, and the whiskers show the ranges.

The vSwitches show an average difference ratio between 1.1$\times$ (vSwitch 4 in Fig. 3) and 5.3$\times$ (vSwitch 3). In addition, the peak difference ratio reaches up to 64.7$\times$ (vSwitch 6). In other words, the control traffic by an SDN controller differs greatly per vSwitch. This is a characteristic of SDN controllers that generate control traffic for each vSwitch depending on the network control [33]–[35]. Such observations necessitate control channel isolation per vSwitch.

D. Control Channel Isolation Problem in SDN-NV

To show the control channel isolation problem in SDN-NV, we measure MTL and NCL by changing the number of vSwitches in a VN between 8, 16, 32, 64, and 128.

1) MTL: We measure three latencies of the network control workflow in SDN-NV—l_V, MTL, and l_P. We measure the latencies for two network controls, forwarding setup and network monitoring. The control messages for the forwarding setup are created as many as the number of vSwitches in a VN in order to install flow rules. Also, the network monitoring generates control messages for statistics requests as the number of vSwitches in a VN. We denote the sum of the three latencies as "total latency," and the total latency is for each control message. Thus, it is different from NCL.



(a) Forwarding setup.

(b) Network monitoring.

Fig. 5: NCL. Details in §II-D2.

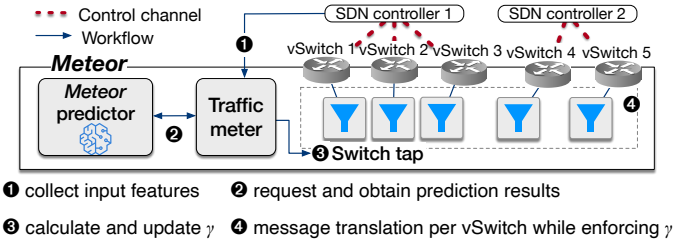


Fig. 6: Architecture and workflow of *Meteor*. Details in §III.

Fig. 4 shows the 99% tail of total latencies that clearly reveals the impact of the control channel isolation.[2] The bars in Fig. 4a represent the total latency of the forwarding setup message for a given number of vSwitches (x-axis). Each bar consists of l_V, MTL, and l_P (e.g., the white portion of each bar is the MTL). In the results, the total latency increases by up to 10.5× as the number of vSwitches increases from 8 to 128. Also, MTL (the white portion of each bar) accounts for 93.3% on average of the total latency, indicating that MTL has the highest impact on total latency. Also, MTL increases by up to 11.7× as the number of vSwitches increases from 8 to 128. On the other hand, l_V and l_P show average latencies of 6.3 ms and 6.4 ms, respectively, without the proportional increase to the number of vSwitches.

Fig. 4b is for network monitoring. The total latency increases by up to 8.6× as the number of vSwitches increases. The MTL also increases by up to 10.1×. Both l_V and l_P show average latencies of 12.7 ms, without the proportional increase to the number of vSwitches. During the experiments (Figs. 4a and 4b), there is no saturation of computing resources, such as CPU, when the vSwitches increase. So, the growth of total latency of a control message only comes from MTL, and it is the main cause that breaks the control channel isolation.

2) NCL: We further investigate the impact of the control channel isolation problem by measuring NCLs for two network controls. The NCL for forwarding setup is the time elapsed from when the first flow rule installation message is generated from an SDN controller to when the last message is delivered to the physical network. Also, NCL for network monitoring is the time elapsed between the statistics requests sent to all the vSwitches and the corresponding replies. Note that the NCL is not just the sum of the total latencies because the SDN controller generates control messages in parallel, and NH also translates the messages in parallel.

We first explain the linear topology results when vSwitches increase from 8 to 128. The results are shown with the line of circle marks in Fig. 5. Fig. 5a shows the NCL for forwarding setup. When the number of vSwitches increases from 8 to 128, NCL increases by up to 12×. Fig. 5b shows the NCL for network monitoring, wherein NCL increases by up to 15×.

[2]We also have checked other percentile values (e.g., 50% median), and the median latencies show similar tendencies to the tail latency.

Next, we additionally measure the NCL[3] on a 4-ary fat-tree topology, which is a realistic topology in datacenters. Compared with the linear topology wherein each switch connects to one or two links, switches in a fat-tree are connected by four links, which is much more complex. The number of physical switches of a 4-ary fat-tree is set to 20. We set the VN topology identical to the physical network; so the number of vSwitches of a VN is also 20. Then, to increase the number of vSwitches, we increase the number of VNs (tenants) from one to six, so the vSwitches increase from 20 to 120. In short, fat-tree topology experiments show NCLs on complex topology with multiple VNs in comparison to the linear.

The lines with × marks in Fig. 5 present the fat-tree topology results. The NCLs of the forwarding setup (Fig. 5a) and network monitoring (Fig. 5b) show quite similar trends and results to those of linear topology, although the fat-tree is complex and involves more VNs (up to 6). The results show that the NCLs and their control channel isolation are related to the number of vSwitches, rather than the topology complexity or the number of VNs.

III. *Meteor* Design

Meteor consists of three components: *Meteor* predictor, traffic meter, and switch tap, as shown in Fig. 6. This section explains the *Meteor* predictor that is an ML model trained offline, followed by the traffic meter that collects input values (features) for the *Meteor* predictor. Then, the switch tap is described, which exists for each vSwitch. To give a big picture, we first outline how *Meteor* components interact as follows.

The traffic meter collects the control traffic and VN topology information within a specific time interval (we call "window") (❶ in Fig. 6). Then, the traffic meter passes them as the input features to the *Meteor* predictor and generates the prediction result for each vSwitch (❷). The prediction result is delivered to the corresponding switch tap. The switch tap calculates the quota γ per vSwitch based on the predicted result (❸). In addition, the switch tap performs message translations while maintaining γ (❹). Note that ❷ and ❸ are repeated at every window. Also, ❶ works periodically (e.g., 100 ms) to collect the features within a window.

[3]We also have measured MTLs on the 4-ary fat-tree topology, but omit the results as they are quite similar to those of linear topology (Fig. 4).

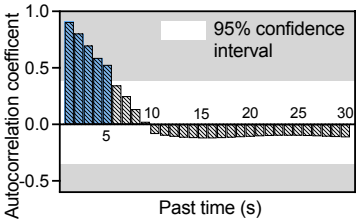


Fig. 7: Autocorrelation coefficient of the control traffic throughput. Details in §III-A1.

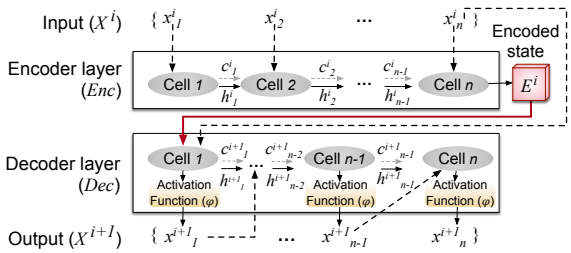


Fig. 8: TapFlow predictor architecture. Details in §III-A2.

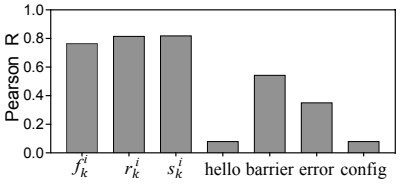


Fig. 9: Correlation analysis between control message types and t_k^i (Pearson R). Details of features (x-axis) in §III-A3.

A. Meteor Predictor

The *Meteor* predictor is designed to predict the time-series of the control traffic throughput. An important design question is which ML model is suitable for *Meteor*. To answer this question, we begin by analyzing the relationship between past and future control traffic. We then describe the structure of the *Meteor* predictor. Finally, we determine the input and output features.

1) Correlation between past and future control traffic: To check the correlation, we use the autocorrelation coefficient that shows the degree of correlation between the data point at the current time with past data points. To collect data, we use the experiment setup in §II-B. We create one VN with varying the number of vSwitches ranging from 1 to 128 and measure the control traffic throughput per second of each vSwitch over 120 s. This experiment is repeated 250 times.

We obtain the 120 autocorrelation coefficients from the past data (e.g., between the most recent one and the past 1 to 120 s). The autocorrelation coefficients are calculated as in [36]. Fig. 7 shows the average autocorrelation coefficients which shows the first 30 s because the coefficients from 31 to 120 s are lower than those of the first 30 s. The x-axis represents the past time relative to the current time, whereby the autocorrelation coefficient is represented as the bar (y-axis). For example, a bar at the 5 s shows the autocorrelation coefficient between the control traffic throughput of the current time (0 s) and of 5 s past. In addition, the region of 95% confidence intervals[4] is shown in white, and the range of higher confidence intervals in gray. Note that if a bar is located in the gray region, the past time of the bar has a strong correlation with the future [36]. In the results, the bars ranging from 1 to 5 s in the past are located in the gray range; thus, the control channel throughput of the past 5 s has a strong correlation with the current control traffic. This implies that it is possible to predict the future control traffic using the past 5 s with good accuracy.

Since the control traffic consists of time-series data, our insight is to use an ML model that utilizes the correlation. To choose the appropriate ML model for the control traffic, we investigate five models that can train on time-series data: 1) ARIMA [37], 2) RNN autoencoder [38], 3) LSTM autoencoder [39], 4) GRU autoencoder [40], and 5) attention-based autoencoder [41]. ARIMA is a traditional modeling method for time-series data. The autoencoder models (four of them) accumulate the past information in so-called "cells." We run separate experiments to choose one out of five models. The results show that the LSTM autoencoder has the best prediction accuracy (see the details in §IV-A1). Thus, we choose the LSTM autoencoder for the *Meteor* predictor.

2) Model structure: The LSTM autoencoder takes the past data and predicts future data. We denote the features of an i-th window as $X^i = \{x_k^i | k = 1, 2, ..., n\}$. Each x_k^i of X^i represents the k-th measured features during the i-th window. We design the *Meteor* predictor to take features of the previous window and to predict the features of the subsequent window. So, the length of the past and future data becomes the window size (e.g., 2 s). The detailed features we select are explained later. Also, n (the number of features in X^i) is determined based on the window size and the measurement interval of the features, which will be also explained later.

Fig. 8 illustrates the structure of the *Meteor* predictor that consists of the encoder layer and decoder layer. The encoder layer (denoted as Enc) compresses the features of the previous window (X^i in Fig. 8) of the control traffic into a vector of fixed dimensionality, called encoded state (E^i) as follows. The creation of E^i is based on the n numbers of LSTM cells in the Enc. Each LSTM cell is designed to learn time-series patterns of X^i. The k-th LSTM cell produces two values: 1) cell state (c_k^i) and 2) hidden state (h_k^i). The first value, c_k^i, delivers the part of accumulated features (i.e., x_1^i to x_k^i), which takes into account n number of LSTM cells (called long-term memory). Also, h_k^i represents the compressed features of x_k^i at a specific LSTM cell (short-term memory).

Specifically, the first LSTM cell takes x_1^i, the first measured features during the window. Then, the cell generates c_1^i and h_1^i. Please refer to [42] for the detailed operations of LSTM cells. The generated c_1^i and h_1^i are fed to the next LSTM cell. From the second LSTM cell, each LSTM cell (such as k-th) takes three values as its inputs: x_k^i, c_{k-1}^i and h_{k-1}^i. Each LSTM cell sequentially produces c_k^i and h_k^i. Finally, h_n^i, which is generated by the last LSTM cell (n-th cell), becomes E^i. This process is formulated as Equation 1.

$$E^i = Enc(x_k^i, c_{k-1}^i, h_{k-1}^i), k = 2, 3, ..., n \quad (1)$$

The decoder layer (Dec) in Fig. 8 predicts the features of the subsequent window (X^{i+1}) as follows. The first LSTM cell of

[4]Note that 95% confidence interval is a de-facto criterion for considering relationships of time-series data [36].

Dec receives the E^i (final hidden state from Enc) and the last x_n^i of input X^i (previous input). Then, the cell generates a new updated hidden state (h_1^{i+1}) from them. Afterward, the cell delivers the h_1^{i+1} to the activation function, φ. On receiving the h_1^{i+1}, φ performs prediction to generate x_1^{i+1}. This prediction is formulated as Equations 2 and 3 and repeated for the n numbers of LSTM cells in Dec. Specifically, the m-th LSTM cell receives the hidden state (h_{m-1}^{i+1}) and the predicted value (x_{m-1}^{i+1}) from the previous cell. Then, the m-th LSTM cell predicts x_m^{i+1}. At the end of the last LSTM cell (i.e., n-th), the prediction is generated by collecting x_m^{i+1} from each cell.

$$h_k^{i+1} = Dec(x_{k-1}^{i+1}, c_{k-1}^{i+1}, h_{k-1}^{i+1}), k = 2, 3, ..., n \quad (2)$$

$$x_k^{i+1} = \varphi(h_k^{i+1}), k = 1, 2, 3, ..., n \quad (3)$$

3) Features: Here, we explain the features (x_k^i) used in the *Meteor* predictor. Note that the input and output features are identical in LSTM autoencoder (§III-A2). Because the *Meteor* predictor is to predict control traffic throughput, we include the control traffic throughput (t_k^i) as input feature in x_k^i. In addition to t_k^i, we consider control message type and data plane as features since they are known to affect t_k^i [33]–[35].

Control message includes several message types: new traffic arrival (f_k^i), flow rule installation (r_k^i), network monitoring (s_k^i), connection establishment (hello), synchronization (barrier), error reporting (error), and device configuration (config) messages. Among the message types, we investigate which one helps improve prediction accuracy by checking the correlation between t_k^i and each message type. We calculate the Pearson R that shows the correlation between two variables [43]. When two variables have a high correlation, the absolute value of the Pearson R approaches 1. Fig. 9 shows seven message types and their Pearson R values. Each bar is the Pearson R value between t_k^i and the number of messages per type (x-axis). Three features—f_k^i, r_k^i, and s_k^i—show notably higher Pearson R values than the others (3.1× higher than those of the others, on average). Thus, we add f_k^i, r_k^i, and s_k^i to the input features ($x_k^i = \{t_k^i, f_k^i, r_k^i, s_k^i\}$).

Second, we include features that represent the data plane. It is known that the data plane (e.g., number of flows, switches, hosts, or ports) impacts the control traffic throughput [33]–[35]. So, we include data plane information related to each vSwitch—the number of hosts attached to the vSwitch (d_k^i) and the number of active links of the vSwitch (a_k^i). Note that we exclude features that are indirectly obtained by features of message types—for example, the number of flows can be derived by f_k^i. Finally, we define six input features for the *Meteor* predictor ($x_k^i = \{t_k^i, f_k^i, r_k^i, s_k^i, d_k^i, a_k^i\}$).

4) Training: The *Meteor* predictor requires a dataset for training. However, to the best of our knowledge, there are no such datasets for SDN systems [44]. Especially, no datasets exist for control traffic in the context of SDN-NV systems. Thus, we generate a dataset to train the *Meteor* predictor. Our dataset is created using the experiment setup of linear topology in §II-B. We randomly select the number of vSwitches from 1 to 128 vSwitches. Note that control channel isolation is highly related to the number of vSwitches regardless of the network topology types (§II-D2); so, we choose the linear topology to reduce the dataset generation complexity. Moreover, we reveal that the *Meteor* predictor trained by the linear topology dataset provides control channel isolation in an even more complex topology (§IV-C).

TABLE I: Representative hyperparameters. Details in §III-A4.

Hyperparameter	Random search range	Best value
# of *Enc* and *Dnc* layers	1–3	2
Hidden state size	16, 32, 64, 128	64
Activation function	Linear, ReLU, leaky ReLU	Linear
Optimizer	ADAM, SGD	ADAM
Batch size	120, 240, 360, 720, 840	720
Learning rate	0.001, 0.003, 0.01, 0.03	0.01
Epoch	100, 200, 400, 1000	200

In addition, for the dataset, the TCP connection between the hosts is generated randomly, which is to simulate the user connections of real systems (e.g., datacenters). The input features are measured every 100 ms. We choose the measurement interval, 100 ms, empirically after numerous internal experiments. We have tested other intervals, but if the intervals become shorter than 100 ms, the NH becomes overloaded and cannot generate the dataset stably. Our SDN controller, ONOS, executes various network controls, such as host location discovery, network topology discovery, forwarding setup, cost-based routing, network monitoring, and switch liveness checking. So, our dataset captures the full spectrum of the network controls.

For the training, we select the hyperparameters of the *Meteor* predictor. We obtain Table I through random searches of 200 trials to determine the best values for the hyperparameters. For the Enc and Dec layers, we stack two layers and set the hidden state size as 64. For the activation function (φ) in Equation 3, we use the linear function [45] to infer the output feature from the last hidden state after we tested other activation functions, such as ReLU and Leaky ReLU. For the optimizer, we test stochastic gradient descent (SGD) and ADAM and choose the ADAM optimizer [46] with a batch size of 720. The learning rate and epoch [47] are configured as 0.01 and 200, respectively.

Lastly, "window size" should be determined. The window size decides the prediction interval of *Meteor*. For example, a window size of 5 s means that the features (X^i) are collected for 5 s, so the prediction is also performed at 5 s. Considering the relationships between the past and future control traffic throughput (Fig. 7), we test the possible window size up to 5 s. Through empirical experiments, we observe that the smaller the window size, the higher the prediction accuracy becomes. However, the NCL improves best when the window size is 2 s due to the operation overheads (§IV-A2). Thus, we set the window size to 2 s. Now the window size is 2 s and, the measurement interval is 100 ms; so, n, the number of elements in X^i, is 20 as 2 s divided by 100 ms.

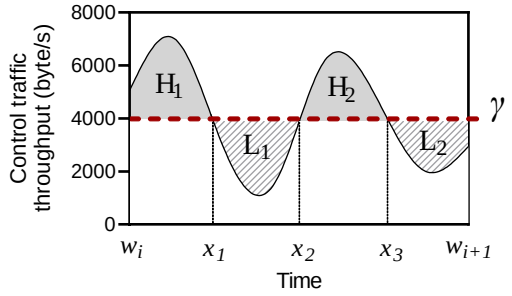


Fig. 10: Control traffic throughput and γ. Details in §III-C1.

B. Traffic Meter

The traffic meter is responsible for two roles—1) measuring and collecting features (❶ in Fig. 6) and 2) requesting and obtaining the prediction results (❷). The traffic meter works in parallel for vSwitches. First, the traffic meter measures the input features of the current window every 100 ms and collects them during the window (2 s) as X^i. The traffic meter feeds X^i to the *Meteor* predictor and receives the prediction result (X^{i+1}). Finally, the traffic meter takes the control traffic throughput (t_1^{i+1} to t_{20}^{i+1}) from the predicted X^{i+1} and delivers it to the switch tap.

C. Switch Tap

The switch tap exists per vSwitch and manages the message translation for the vSwitch. It performs two operations: 1) γ calculation (❸ in Fig. 6) and 2) γ enforcement (❹).

1) γ calculation: When the switch tap receives the predicted control traffic throughput for the subsequent window (t_1^{i+1} to t_{20}^{i+1}), it calculates γ as in the following example. Let the graph in Fig. 10 be $P_i(t)$ that is a function of the predicted control traffic throughput over time. Suppose that the dotted line in Fig. 10 (4000 bytes/s) represents γ. The time w_i is the start time of the i-th window. Also, the region "H" (i.e., H_1 and H_2) indicates the time periods when the control traffic throughput is higher than γ, and "L" (i.e., L_1 and L_2) for lower throughput. For the time period of H_1 (w_i to x_1 in the x-axis of the graph), the switch tap stores the excess control traffic in its buffer. Then, from x_1 to x_2 (L_1 region), where its throughput is less than γ, the switch tap processes the buffered traffic. Similarly, the excess traffic buffered during H_2 gets translated during L_2.

If H regions are bigger than L regions, the buffered traffic is not translated during x_1 to x_2 and x_3 to w_{i+1}. The remainder "H–L" will be processed after w_{i+1}, which delays the message translation. The opposite case is when L is bigger than H, meaning that γ is over-reserved for the vSwitch. Based on these observations, we derive the most effective γ_i (γ of the i-th window) when H = L. So, the switch tap calculates γ using Equation 4 where $|W|$ is the window size. In this way, message translation is performed with γ_i during the i-th window, even when the control traffic generation exhibits burstiness. This contributes to enhancing the control channel isolation.

$$\int_{w_i}^{w_{i+1}} P_i(t)\,dt = |W| \times \gamma_i \tag{4}$$

2) γ enforcement: Now with γ_i, *Meteor* enforces it as follows. The switch tap does bookkeeping whenever it translates messages and calculates the average amount of translated control traffic per second (m_{tr}). For new control messages, it checks whether the m_{tr} exceeds γ_i. If yes, the switch tap stops the message translation. If not, it performs message translation. While the message translation is stopped, the m_{tr} decreases. Also, if there is any untranslated control traffic when the message translation is stopped, the untranslated traffic is stored in the buffer of the switch tap (§III-C1). Afterward, when the m_{tr} goes below γ_i, the switch tap resumes the message translation. The switch tap then translates the buffered traffic and the newly arriving traffic.

A remaining question is how often the switch tap checks whether the m_{tr} exceeds γ_i. If the checking interval is short, the switch tap can stop and resume message translations frequently, so the γ enforcement can be more accurate. However, short intervals can result in high overheads to *Meteor* (e.g., CPU utilization). Through experiments, we empirically find that 100 ms is appropriate for *Meteor*.

IV. Implementation and Evaluation

We implement the traffic meter and switch tap within Libera, an up-to-date open-source NH (3.2K LoC). For the *Meteor* predictor, we use PyTorch v1.10.1 to implement the LSTM autoencoder (0.7K LoC). The *Meteor* predictor is trained by a dataset of 360K records. We empirically observe that the dataset of 360K records achieves a reasonable prediction accuracy to be shown in §IV-A1. Also, we train the *Meteor* predictor offline because the offline trained model predicts well even when the control traffic and topology are not observed in the training dataset (§IV-C). The model takes 4.5 hours for its training. Our implementation and execution instructions are accessible on [48].

With the implementation, this section evaluates *Meteor*. All experiments are conducted with the experiment setup in §II-B. Note that an NVIDIA RTX 2080 Ti GPU is additionally used to train the *Meteor* predictor. We conduct four sets of experiments. The first set is to evaluate how accurate the *Meteor* predictor is. The second set is to evaluate how effectively *Meteor* isolates the control channels. The third set evaluates whether *Meteor* can be generalizable to predict the control traffic of different topologies and multiple tenants. The fourth set is to evaluate the overheads of *Meteor*. Each set is explained in its corresponding subsection. All experiments last for two minutes and are repeated five times.

A. Prediction Accuracy

Here, we evaluate the prediction accuracy of the *Meteor* predictor in two directions: 1) model structure design and 2) window size. First, regarding the model structure, we compare the *Meteor* predictor with other ML models by measuring

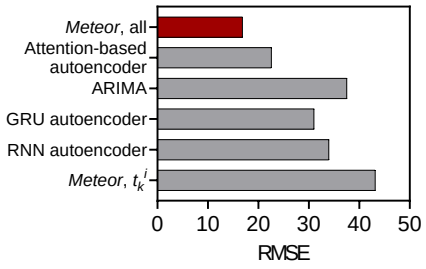


Fig. 11: RMSE comparison. Details in §IV-A1.

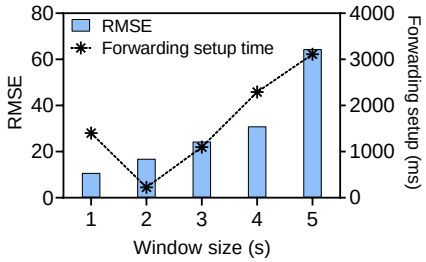
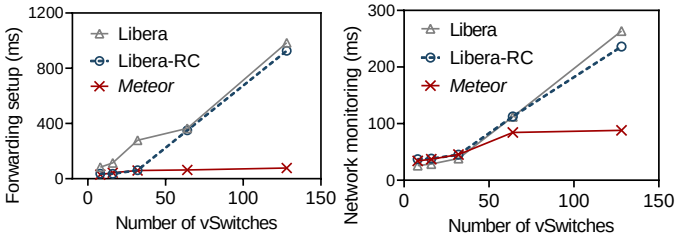
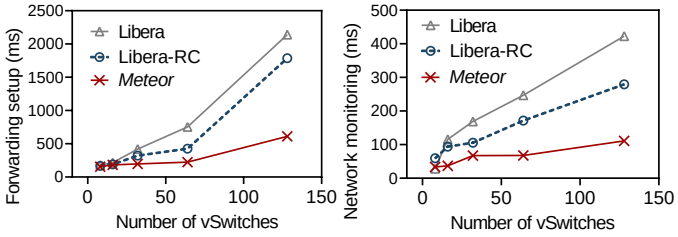


Fig. 12: RMSE and forwarding setup time per window size. Details in §IV-A2.



(a) Forwarding setup. (b) Network monitoring.

Fig. 13: MTL comparison. Details in §IV-B1.



(a) Forwarding setup. (b) Network monitoring.

Fig. 14: NCL comparison. Details in §IV-B2.

the prediction accuracy for future control traffic throughput. The accuracy is calculated as the root mean squared error (RMSE), which is commonly used to evaluate the prediction accuracy of time-series data [37]. We measure the accuracy with a new dataset containing 10K records, not used to train the *Meteor* predictor. Second, for the window size, we validate the decision rationale of 2 s window in *Meteor* in terms of the prediction accuracy and NCL.

1) Model accuracy: Fig. 11 shows the prediction error (RMSE) of the *Meteor* predictor with the other models. We first compare the *Meteor* predictor (the first bar in Fig. 11) to four other ML models (i.e., attention autoencoder (AE), ARIMA, GRU AE, and RNN AE). All the models in Fig. 11 are trained by the same dataset and identical input features (i.e., t_k^i, f_k^i, r_k^i, s_k^i, d_k^i, and a_k^i). The hyperparameters of the models are found in the same manner as in §III-A4. The results in Fig. 11 shows that the *Meteor* predictor outperforms the other models by 1.3×, 2.2×, 1.8×, and 2× better accuracy.

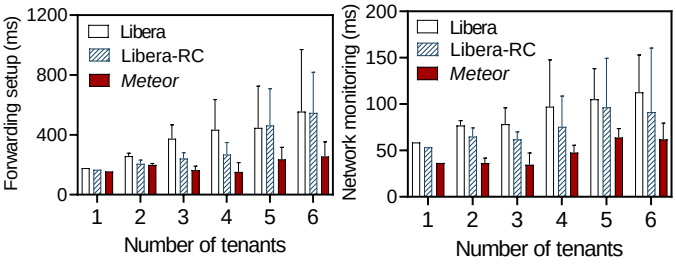
In addition, we check the effect of input features by comparing when only t_k^i is used as features (the last bar in Fig. 11) and when all features (i.e., t_k^i, f_k^i, r_k^i, s_k^i, d_k^i, and a_k^i) are used. The first bar represents the prediction error of *Meteor* with the six input features. In the results, the first bar exhibits 2.6× better prediction accuracy than the last bar. The results validate our selection of input features.

2) Effect of window size: We vary the window size from 1 s to 5 s. Over the window size, we measure the prediction accuracy of the *Meteor* predictor and the NCL. The NCL is measured on a VN of 64 vSwitches (linear topology) with the network control of forwarding setup. The results are in Fig. 12 where the prediction accuracy is shown with bars, and NCL is shown by a dotted line. In terms of prediction accuracy, the shortest window size (1 s) demonstrates the best prediction—the RMSE becomes 6× better (smaller) than when the window size is 5 s. The prediction errors increase gradually from 1 s to 4 s and then increase rapidly when the window size becomes 5 s by 2.1×.

However, the NCL for forwarding setup shows different results. From window sizes of 2 s to 5 s, the NCL increases as the window size (up to 13.9×). Surprisingly, NCL is smallest with 2 s, and, when the window size changes to 1 s, the NCL rather increases by 6.2×. We find that this is due to the overhead of *Meteor*. *Meteor* predicts the control traffic and calculates γ per window for every vSwitch; thus, if the window size is too small, the *Meteor* operations (Fig. 6) are executed too often, becoming a bottleneck. Thus, we choose 2 s as the optimal window size that balances the prediction accuracy and NCL. We run more experiments with a larger number of vSwitches and NCL for other network controls (e.g., network monitoring) and find that the results have a similar tendency.

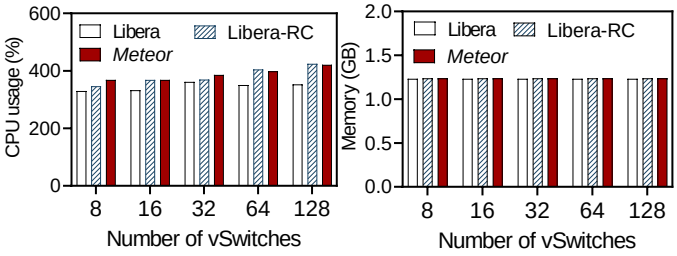
B. Control Channel Isolation

Here, we evaluate the improvement in control channel isolation when the number of vSwitches increases from 8 to 128. Two metrics are used—MTL and NCL—for forwarding setup and network monitoring. We compare *Meteor* with two other NHs: Libera [5] and Libera-RC. Libera does not offer control channel isolation, so we implement Libera-RC that calculates γ based on reactive control [49], [50], which does not use any predictions. Specifically, Libera-RC works as follows. At first, each vSwitch has identical γ as its initial value. Then, Libera-RC checks the control traffic amount during the past 2 s of each vSwitch. If the amount is higher than the γ value, it increases γ for a certain percentage. On the other hand, if lower, it decreases γ. Libera-RC implementation updates γ by incrementing or decrementing the past γ by 5%. We set



(a) Forwarding setup. (b) Network monitoring.

Fig. 15: *Meteor* generalizability—NCL comparison by multiple tenants over a 4-ary fat-tree topology. Details in §IV-C.



(a) Memory consumption. (b) CPU cycles.

Fig. 16: Overheads. Details in §IV-D.

5% empirically to get the best results from Libera-RC. We use Libera-RC to see the effectiveness of the prediction of *Meteor*.

1) MTL: We measure MTL per each control message and show the 99% tail MTL. Fig. 13a shows the MTL for the network control of forwarding setup. For the small number of vSwitches, such as 8, three NHs show relatively similar MTLs, such as 50.1 ms on average. However, as the number of vSwitches increases to 128, MTLs diverge significantly. For 128 vSwitches, *Meteor* reduces MTL up to 92.2% and 91.7% (12.7× and 12× improvements), respectively, compared to Libera and Libera-RC. Second, Fig. 13b shows the MTL of network monitoring. The MTLs of Libera and Libera-RC continuously increase by up to 10.1× and 6.4×, respectively. On the other hand, the MTL of *Meteor* increases only up to 2.6×, which is the 3.9× and 2.5× improvements over Libera and Libera-RC.

2) NCL: Fig. 14a shows the average NCL for forwarding setup. All NHs have similar NCL with 8 vSwitches—166.6 ms on average. However, when the number of vSwitches increases to 128, the NCLs of Libera, Libera-RC, and *Meteor* increase up to 12×, 10.7×, and 3.9×, respectively. This means that *Meteor* helps a tenant finish the forwarding setup up to 3.5× and 2.9× faster (71.4% and 65.7% shorter) than Libera and Libera-RC. Fig. 14b depicts the average NCL for network monitoring. For 8 vSwitches, again, there is a little difference in NCL between three NHs—40.4 ms on average. However, as the vSwitches increase, their differences become greater. For example, with 128 vSwitches, *Meteor* enables the network monitoring 3.8× and 2.5× faster (73.7% and 60.2% shorter) than Libera and Libera-RC, respectively. Note that NCLs in non-virtualized SDN reach up to 1 s for stress tests in the control channel [22]–[24]. *Meteor* exhibits NCLs by up to 612 ms (128 vSwitches in Fig. 14a), which means that *Meteor* is as efficient as the non-virtualized SDN by achieving the control channel isolation.

C. Generalizability

In this subsection, we evaluate whether *Meteor* can be applicable with network topologies and VN settings different from those used in the training dataset—we call it generalizability. Note that the training dataset of *Meteor* is generated in a linear topology with one tenant. To demonstrate the generalizability, we experiment with 6 tenants in 4-ary fat-tree topology (20 physical switches). Fat-tree is the representative topology for datacenters, and it is much more complex than the linear topology as it has multiple forwarding paths between hosts. For experiments, each VN is created as a tree topology containing 10 vSwitches, so the number of vSwitches increases from 10 to 60. Then, the NCLs for forwarding setup and network monitoring are measured. We also measure MTL, but the results are not included in this paper due to the space limit.

1) NCL for forwarding setup: Fig. 15a shows the average NCL for forwarding setup as the number of tenants increases. The x-axis is the number of tenants, and the y-axis is the NCL with the whiskers of NCL ranges. As the number of tenants increases from 1 to 6, the average forwarding setup times of Libera, Libera-RC, and *Meteor* increase 3.12×, 3.28×, and 1.65×, respectively. So, when the number of tenants is 6, *Meteor* improves (reduces) NCL up to 2.2× and 2.1× than Libera and Libera-RC, respectively. These improvements for forwarding setup are less than the improvements of *Meteor* on the experiment setup used for the training dataset (i.e., 3.5× and 2.9×), but it still achieves reasonable improvements over more complex topologies and higher numbers of tenants.

2) NCL for network monitoring: We evaluate the generalizability with network monitoring. Fig. 15b shows the average NCL for network monitoring. *Meteor* always shows the smallest latency—on average, 46.2% and 35.6% lower than Libera and Libera-RC. In addition, *Meteor* exhibits a maximum of 2.3× and 1.8× improved (reduced) latency than Libera and Libera-RC, respectively. Similar to the forwarding setup, NCL for network monitoring improves less than the experiment setup used for the training dataset (i.e., 3.8× and 2.5×). However, *Meteor* still shows significant improvements over a complex experiment setup. We believe that the results confirm the generalizability of *Meteor*.

D. Overheads

We evaluate the overhead of *Meteor* in terms of memory consumption and CPU cycles. In the experiment setup (§II-B), the memory consumption and CPU cycles are measured in a server that runs *Meteor* (i.e., Intel Xeon E5-2600 CPUs and 64 GB memory). We vary the number of vSwitches from 8 to 128. Fig. 16 shows the measurement results. The memory consumption is represented by bars in Fig. 16a. Compared to

Libera, *Meteor* consumes 5.3 MB higher memory on average, which is only 0.4% increase.

The CPU cycles are shown in Fig. 16b. Note that the CPU cycles cover the entire workflow of *Meteor*, but not the *Meteor* predictor because a GPU (NVIDIA RTX 2080 Ti) is used. Fig. 16 shows that, on average, *Meteor* consumes 12% more CPU cycles than Libera and 1% more than Libera-RC. This increase is because *Meteor* collects input features for each vSwitch, and calculates and enforces γ per window. Given that *Meteor* yield up to 3.8× improvement in NCL (§IV-B2), we believe that the increase in CPU cycles is acceptable.

V. Discussion

***Meteor* predictor for other SDN controllers.** The *Meteor* predictor in this study is trained based on the dataset of ONOS controller. A question is whether the *Meteor* predictor can be used for other SDN controllers. We test its feasibility by training the *Meteor* predictor for a different SDN controller, i.e., OpenDaylight (ODL) [51], which is another well-known controller. We generate a dataset for ODL with the same setup of ONOS (§III-A4). The newly trained *Meteor* predictor for ODL shows an RMSE of 19.8, which is quite comparable to that of ONOS (i.e., 16.9 in Fig. 11). The results show that the *Meteor* predictor proposed in this study can be trained on other SDN controllers with reasonable accuracy; so, we believe that the *Meteor* predictor can be used with other SDN controllers.

Necessity of prediction. In this study, we use prediction to offer control channel isolation in SDN-NV. One might be curious why the prediction is required. The reason we use the prediction is that each vSwitch receives greatly different control traffic in bursts; thus, it is difficult to calculate the proper γ without predicting the future control traffic. Indeed, we have shown that the γ calculation only based on past information (e.g., Libera-RC) is insufficient in offering control channel isolation. For example, in MTL, as shown in Fig. 13a, Libera-RC shows little difference (i.e., 4.3% differences on average) from Libera when the number of switches increases to 64 and 128. Thus, we believe the prediction in *Meteor* is a proper design choice to offer control channel isolation.

Global γ calculation. *Meteor* calculates γ per vSwitch individually. One possible question is the necessity of a global γ calculation by considering all the vSwitches together. The global γ calculation might be necessary to provide fairness among vSwitches. However, an SDN controller creates control traffic per vSwitch, which, by nature, greatly differs as discussed in §II-C2. So, the fairness itself is controversial because it could hurt the end-to-end network control from the SDN controller. In addition, the global calculation causes non-trivial overheads because it requires periodic synchronization between vSwitches to manage all the vSwitches globally. Thus, we leave this topic for future research.

Using different window sizes. It is possible that the window size can be different for input (X^i) and output (X^{i+1}) features. For example, predicting the next 2 s by considering the past 5 s is a possibility. Most existing studies in ML fields, such as natural language and speech processing, use LSTM autoencoder with the same window size for X^i and X^{i+1} [39], [52]. To verify this possibility, however, we run additional experiments that are not reported in this paper. We find that *Meteor* shows 19× higher prediction accuracy when the window sizes of X^i and X^{i+1} are the same (i.e., 2 s), comparing the case of the different sizes (i.e., 5 s and 2 s as X^i and X^{i+1}, respectively). Thus, we set the same window size for the input and output features.

VI. Related Work

NH. The studies such as FlowVisor [20], FlowN [17], and OpenVirteX [16], defined and introduced key message translation operations such as topology and address. CoVisor [15] proposed a message translation to composite multiple flow rules in an NH that enabled multiple SDN controllers to manage a single network. Libera [5] defined a cloud service model, called network infrastructure-as-a-service, and summarized key message translation techniques to improve the scalability and flexibility of an NH. V-Sight [11] and TeaVisor [19] further developed message translation in NHs to achieve accurate network monitoring and bandwidth isolation for the data plane, respectively. Furthermore, HyperFlex [53] sliced the CPU of the NH by de-compositing the monolithic NH into micro functions. However, the above studies lack a control channel isolation scheme.

ML on SDN systems. As ML becomes common tools for making optimized decisions, many applications on SDN systems, such as routing optimization, security, QoS prediction, traffic classification, and resource management, have been using ML (e.g., random forest, neural network, and reinforcement learning) [44]. However, ML has not been used for control channel isolation, which is the focus of our study.

VII. Conclusion

This study presents *Meteor* that offers the control channel isolation in SDN-NV by predicting the control traffic. *Meteor* designs an LSTM autoencoder to calculate γ and enforces γ per virtual switch. The performance evaluation of *Meteor* reveals that MTL is enhanced up to 12.7×, which improves the control channel isolation significantly. In addition, the NCL reduces by up to 73.7% over the existing NHs without prediction. Furthermore, we show that *Meteor* is quite generalizable in that it achieves reasonable improvements even when it runs in different topologies from the training dataset. As these results make the end-to-end control latency comparable to the non-virtualized SDN, we hope that this study contributes to ushering in the practical use of SDN-NV in datacenters.

Acknowledgment

This work was supported in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2021R1A6A1A13044830), by the NRF grant funded by the Korea government (MSIT) (NRF-2023R1A2C3004145), and by a Korea University Grant. Co-corresponding authors are Gyeongsik Yang and Chuck Yoo.

REFERENCES

[1] D. Firestone, "VFP: A virtual switch platform for host SDN in the public cloud," in *14th USENIX Conference on Networked Systems Design and Implementation*, 2017, p. 315–328.

[2] A. Roy, D. Bansal, D. Brumley, H. K. Chandrappa, P. Sharma, R. Tewari, B. Arzani, and A. C. Snoeren, "Cloud datacenter SDN monitoring: Experiences and challenges," in *Internet Measurement Conference 2018*. ACM, 2018.

[3] T. Koponen *et al.*, "Network virtualization in multi-tenant datacenters," in *11th USENIX Conference on Networked Systems Design and Implementation*, 2014, p. 203–216.

[4] Y. Yoo, G. Yang, J. Lee, C. Shin, H. Kim, and C. Yoo, "TeaVisor: Network hypervisor for bandwidth isolation in SDN-NV," *IEEE Transactions on Cloud Computing*, 2022.

[5] G. Yang, B.-y. Yu, H. Jin, and C. Yoo, "Libera for programmable network virtualization," *IEEE Communications Magazine*, vol. 58, no. 4, pp. 38–44, 2020.

[6] G. Yang, C. Shin, Y. Yoo, and C. Yoo, "A case for SDN-based network virtualization," in *29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2021.

[7] L. Bonati, M. Polese, S. D'Oro, S. Basagni, and T. Melodia, "Open, programmable, and virtualized 5g networks: State-of-the-art and the road ahead," *Computer Networks*, vol. 182, 2020.

[8] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca, and J. Folgueira, "Network slicing for 5G with SDN/NFV: Concepts, architectures, and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 80–87, 2017.

[9] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "ATP: In-network aggregation for multi-tenant learning," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX, 2021, pp. 741–761.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *ACM SIGCOMM 2011 Conference*. ACM, 2011, p. 254–265.

[11] G. Yang, H. Jin, M. Kang, G. Jun Moon, and C. Yoo, "Network monitoring for SDN virtual networks," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pp. 1261–1270.

[12] H. Jin, G. Yang, B.-y. Yu, and C. Yoo, "TALON: Tenant throughput allocation through traffic load-balancing in virtualized software-defined networks," in *International Conference on Information Networking (ICOIN)*, 2019, pp. 233–238.

[13] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling distributed machine learning with In-Network aggregation," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX, pp. 785–808.

[14] C. Shin, G. Yang, Y. Yoo, J. Lee, and C. Yoo, "Xonar: Profiling-based job orderer for distributed deep learning," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, 2022, pp. 112–114.

[15] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A compositional hypervisor for Software-Defined networks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX, 2015, pp. 87–101.

[16] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "OpenVirteX: Make your virtual SDNs programmable," in *Third Workshop on Hot Topics in Software Defined Networking*. ACM, 2014, p. 25–30.

[17] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *IEEE Internet Computing*, vol. 17, no. 2, pp. 20–27, 2013.

[18] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *13th Usenix Conference on Networked Systems Design and Implementation*, 2016, p. 407–424.

[19] G. Yang, Y. Yoo, M. Kang, H. Jin, and C. Yoo, "Bandwidth isolation guarantee for SDN virtual networks," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10.

[20] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *9th USENIX Conference on Operating Systems Design and Implementation*. USENIX, 2010, p. 365–378.

[21] M. Dalton *et al.*, "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *15th USENIX Symposium on Networked Systems Design and Implementation*, 2018, pp. 373–387.

[22] M. Karakus and A. Durresi, "A survey: Control plane scalability issues and approaches in software-defined networking (SDN)," *Computer Networks*, vol. 112, pp. 279–293, 2017.

[23] L. Zhu, M. M. Karim, K. Sharif, C. Xu, F. Li, X. Du, and M. Guizani, "SDN controllers: A comprehensive analysis and performance evaluation study," *ACM Comput. Surv.*, vol. 53, no. 6, 2020.

[24] A. Nguyen-Ngoc, S. Lange, S. Geissler, T. Zinner, and P. Tran-Gia, "Estimating the flow rule installation time of sdn switches when facing control plane delay," in *International Conference on Measurement, Modelling and Evaluation of Computing Systems*. Springer, 2018, pp. 113–126.

[25] P.-W. Tsai, C.-W. Tsai, C.-W. Hsu, and C.-S. Yang, "Network monitoring in software-defined networking: A review," *IEEE Systems Journal*, vol. 12, no. 4, pp. 3958–3969, 2018.

[26] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," in *ACM SIGCOMM 2010 Conference*, 2010.

[27] M. Rifai, N. Huin, C. Caillouet, F. Giroire, D. Lopez-Pacheco, J. Moulierac, and G. Urvoy-Keller, "Too many SDN rules? compress them with MINNIE," in *2015 IEEE Global Communications Conference*, 2015, pp. 1–7.

[28] R. Khalili, W. Y. Poe, Z. Despotovic, and A. Hecker, "Reducing state of openflow switches in mobile core networks by flow rule aggregation," in *25th International Conference on Computer Communication and Networks (ICCCN)*, 2016, pp. 1–9.

[29] Q. T. Minh, A. Van Le, T. K. Dang, T. Nam, and T. Kitahara, "Flow aggregation for sdn-based delay-insensitive traffic control in mobile core networks," *IET Communications*, vol. 13, no. 8, pp. 1051–1060, 2019.

[30] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in Software-Defined networks," in *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. USENIX, 2012.

[31] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010.

[32] "Open Network Operating System (ONOS)," https://opennetworking.org/onos/, Accessed: 2022-11-03.

[33] A. Bianco, P. Giaccone, R. Mashayekhi, M. Ullio, and V. Vercellone, "Scalability of ONOS reactive forwarding applications in ISP networks," *Computer Communications*, vol. 102, 2017.

[34] A. Bianco, P. Giaccone, A. Mahmood, M. Ullio, and V. Vercellone, "Evaluating the SDN control traffic in large ISP networks," in *2015 IEEE International Conference on Communications (ICC)*, 2015.

[35] B.-y. Yu, G. Yang, and C. Yoo, "Comprehensive prediction models of control traffic for SDN controllers," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, pp. 262–266.

[36] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[37] S. Siami-Namini, N. Tavakoli, and A. Siami Namin, "A comparison of ARIMA and LSTM in forecasting time series," in *2018 17th IEEE International Conference on Machine Learning and Applications)*, 2018.

[38] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *2014 Conference on Empirical Methods in Natural Language Processing*. ACL, 2014.

[39] V. Wan, Y. Agiomyrgiannakis, H. Silen, and J. Vit, "Google's next-generation real-time unit-selection synthesizer using sequence-to-sequence LSTM-based autoencoders," in *INTERSPEECH*, 2017.

[40] Y. Guo, W. Liao, Q. Wang, L. Yu, T. Ji, and P. Li, "Multidimensional time series anomaly detection: A GRU-based gaussian mixture variational autoencoder approach," in *10th Asian Conference on Machine Learning*, vol. 95. PMLR, 2018, pp. 97–112.

[41] Y. Qin, D. Song, H. Cheng, W. Cheng, G. Jiang, and G. W. Cottrell, "A dual-stage attention-based recurrent neural network for time series prediction," in *26th International Joint Conference on Artificial Intelligence*. AAAI Press, 2017, p. 2627–2633.

[42] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[43] J. Adler and I. Parmryd, "Quantifying colocalization by correlation: The Pearson correlation coefficient is superior to the Mander's overlap coefficient," *Cytometry Part A*, vol. 77A, no. 8, pp. 733–742, 2010.

[44] J. Xie, F. R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, and Y. Liu, "A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges," *IEEE Communications Surveys Tutorials*, pp. 393–430, 2019.
[45] S. Qian, H. Liu, C. Liu, S. Wu, and H. S. Wong, "Adaptive activation functions in convolutional neural networks," *Neurocomputing*, vol. 272, pp. 204–212, 2018.
[46] I. Bello, B. Zoph, V. Vasudevan, and Q. V. Le, "Neural optimizer search with reinforcement learning," in *34th International Conference on Machine Learning*, vol. 70, 2017, p. 459–468.
[47] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems*, vol. 24, 2011.
[48] Y. Yoo, G. Yang, C. Shin, and J. Lee, "yeonhooy/Meteor_Control_Channel_Isolation_SDN_Virtualization: v1.3," Feb. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7627685
[49] R. Arkin, "Reactive control as a substrate for telerobotic systems," *IEEE Aerospace and Electronic Systems Magazine*, pp. 24–31, 1991.
[50] A. R. Aron, "From reactive to proactive and selective control: Developing a richer model for stopping inappropriate responses," *Biological Psychiatry*, vol. 69, no. 12, pp. e55–e68, 2011.
[51] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a model-driven SDN controller architecture," in *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, 2014, pp. 1–6.
[52] J. Li, M.-T. Luong, and D. Jurafsky, "A hierarchical neural autoencoder for paragraphs and documents," in *ACL (1)*, 2015, pp. 1106–1115.
[53] A. Blenk, A. Basta, and W. Kellerer, "HyperFlex: An SDN virtualization architecture with flexible hypervisor function allocation," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015, pp. 397–405.