STEREOS: Smart Table EntRy Eviction for OpenFlow Switches

Hemin Yang, George F. Riley, and Douglas M. Blough

***Abstract*— Software-defined networking (SDN) is fundamentally changing the way networks operate, enabling programmable and flexible network management and configuration. As the *de facto* standard southbound interface of SDN, OpenFlow defines how the control plane interacts with the data forwarding plane. In OpenFlow, flow tables play a significant role in packet forwarding. However, the size of the flow table is limited due to power, cost, and silicon area constraints and capacity-limited tables cannot hold all of the active flows in medium-to-large-scale SDN networks. Thus, when a flow table reaches capacity, an intelligent eviction strategy, which efficiently manages the limited flow table resource, is critical. In this paper, we propose Smart Table EntRy Eviction for OpenFlow Switches (STEREOS), which uses machine learning to classify flow entries as active or inactive and forms the basis for intelligent eviction. Trace-driven simulations demonstrate that STEREOS increases flow table usage by more than 50% and reduces incorrect flow entry evictions by up to 78%, compared with the dominant Least Recently Used eviction policy. Moreover, packet-level simulations of a datacenter network demonstrate that STEREOS can greatly reduce the control overhead, increase overall network throughput by 19%, and reduce packet loss rate by 70%.**

***Index Terms*— Flow table eviction, machine learning, OpenFlow switch, software-defined networks.**

I. Introduction

SOFTWARE Defined Networking (SDN) is widely regarded as a revolutionary technology due to its capability to create programmable, flexible and agile networks while reducing costs. Google, Amazon, Facebook, and other organizations have heavily researched and invested in SDN for their data center networks. For example, Google leveraged SDN to build its Jupiter network, which achieved a capacity increase of 100× [1], and Microsoft used SDN to improve its hyperscale data center connectivity, achieving substantial reductions in capital and operating expenses [2].

The core idea of SDN is to separate the control plane from the forwarding/data plane in switches. This makes network operations programmable and accelerates innovations through SDN abstractions. Most SDN implementations use the OpenFlow protocol [3] as the communication interface between control and data planes. The kernel of OpenFlow is a packet processing pipeline consisting of several flow tables, which contain flow entries that are used to match and process incoming packets. Due to power, cost, and silicon area constraints, the flow table size is a limitation for networks with a large number of flows. As reported by Lu *et al.* [4], the Broadcom chipset, which is widely used in commercial switches, can accommodate 2000 flow entries. On the other hand, flow arrival rates can reach 10,000 flows per second per server rack in data centers [5]. Hence, it is extremely important to manage flow tables efficiently, and the primary tool for this is accurate identification of inactive flows when a flow table has reached capacity and an existing flow must be evicted.

In this paper, we propose a new approach to flow entry eviction, which is based on machine learning. Patterns of active and inactive flows in a particular network environment are learned in a training phase and the learned models are then applied to the operational flows in the network. Based on multiple network traces from different real networks, we study which machine learning models are well suited for this problem and we demonstrate the efficacy of those models in the associated networks. The paper's contributions include:

1) identification of specific features of flows that are important for active/inactive flow classification,
2) the Smart Table EntRy Eviction for OpenFlow Switches (STEREOS) prediction mechanism, which trains an offline machine learning (ML) model to predict the probability that a flow entry is inactive; this can be used by OpenFlow switches to more accurately evict inactive flow entries once a flow table has reached capacity, and
3) detailed case studies that address practical problems in the implementation of STEREOS such as ML model selection, ML model size trade-off, overhead, and feature quantization, which provide a roadmap for implementation of STEREOS in real SDN networks.

II. Related Work

Machine learning is a promising approach to optimize network operations. Clark *et al.* [6], proposed to introduce a knowledge plane into the Internet to enable machine learning applications 16 years ago but knowledge plane prototypes have still not appeared, due mainly to the decentralized Internet

Manuscript received June 20, 2019; revised October 14, 2019; accepted November 6, 2019. Date of publication December 12, 2019; date of current version February 19, 2020. The article was presented at the IEEE ICCCN 2018. *(Corresponding author: Hemin Yang.)*

H. Yang was with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA. He is now with Microsoft, Redmond, WA, USA.

G. F. Riley, deceased, was with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA.

D. M. Blough is with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: doug.blough@ece.gatech.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2019.2959184

control structure. However, SDN networks, have centralized network controllers and typically operate at the level of a single organization, which facilitates collection of the data necessary to train machine learning models. To date, machine learning techniques have been applied to address several networking problems in SDNs, e.g. routing optimization [7], [8], management of resources such as network bandwidth and compute capacity [9], [10], and security [11], [12].

The specific problem studied herein, flow entry eviction, is one aspect of flow table management in SDNs. OpenFlow currently provides three mechanisms for flow table management: flow expiration timeouts, proactive flow entry deletion, and flow entry eviction [3]. To date, only a few papers have studied machine-learning-based flow table management. Li *et al.* [13], proposed to use Q-learning for the selection of flow expiration timeout values. Yang and Riley [14] employed machine learning to learn from historical flow entry removal statistics how to predict when a flow entry will be last used. As for flow entry eviction, Kannan and Banerjee [15], built a Markov based learning predictor that captures the probability of transitioning between different intervals and evicts the flow entry in the state from which the transition probability is the least. However, their approach assumes that flow arrivals follow a Poisson distribution, which is not necessarily true in practice. This paper is a significant extension of our prior work in which we first proposed network-data-based machine learning for switch-controlled on-demand flow entry eviction [16].

Most prior work on switch-controlled on-demand flow entry eviction has studied traditional (not machine learning) approaches, e.g. [17]–[19]. Both [18] and [19] are based on categorizing flows as "elephant" flows, which are small in number but represent a large fraction of traffic volume, and "mouse" flows, which are high in number but carry little data. The goal for both of these works is to reduce the number of elephant flow evictions. The goal of our approach, and also that of [17], is simply to minimize flow table misses, regardless of flow type, since each flow miss causes an expensive switch to controller communication in order to retrieve the appropriate handling rule for the missing flow. Whereas [17] uses an approach based on Bloom filters, ours is based on learning the traffic patterns in a particular network context.

III. Smart Flow Table Entry Eviction

If an active flow entry is wrongly evicted from a flow table, the switch has to query the controller to reinstall the flow entry when the packets of the flow arrive in the future. This re-installation not only incurs additional delays [20] but also increases the controller's workload. Furthermore, evicting an active TCP flow entry can seriously degrade the performance of TCP connections, because it may result in packet loss and congestion window shrinkage for all TCP flows that share the same switch buffer [21]. Therefore, minimizing incorrect flow entry evictions is critical to maintaining network performance.

We approach this problem by considering how to train a binary classification model that can label each flow as either active or inactive. Our proposed approach, referred to as STEREOS, is shown in Fig. 1. The first phase is offline training to generate the classification model, and the second phase is run-time use of the trained model to perform flow table eviction. To do the offline training, packet traces from a few OpenFlow switches in the target network are captured with a tool such as Wireshark. These traces are used to extract features and labels to form the training dataset. The dataset is then used to train the offline model including tuning its hyperparameters. To implement STEREOS in a real network, the parameters of the trained model would then be distributed to the OpenFlow switches by the controller. Lastly, every OpenFlow switch would apply the trained model to identify and evict inactive flow entries. The model training could be performed by a helper device attached to the network [5] or by the controller itself.

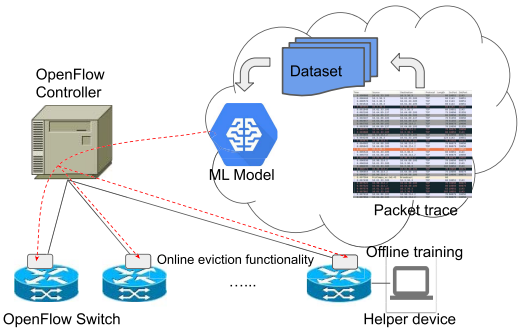

Fig. 1. Overview of STEREOS.

Deploying an improved eviction algorithm will require that SDN switches be modified, because current switches use LRU replacement as the default. Machine learning models of the type considered herein have been shown to have efficient FPGA implementations [22], [23], and our results show that these models provide substantial benefits over LRU across a range of scenarios. Thus, these results provide motivation to designers of future switches to consider implementing more advanced eviction algorithms such as those proposed herein.

Offline training with online eviction can be periodically updated (e.g., every 10 hours) in order to adapt to changes in the underlying traffic pattern. Here, we do not use online classification because we can only get ground truth labels (inactive or active) through a long time observation (at least tens of minutes). Our trace-driven evaluations reported in Section IV indicate that flow characteristics in data center environments do not change rapidly making the offline training approach feasible. For example, with one set of traces covering a 22-hour period, we trained a model based on the first two hours of the traces and then very accurately identified inactive flows during the remaining 20-hour duration.

A. Offline Training: Data Collection

In order to train a model for classifying flow table entries as active or inactive, we need training data on flows. Every data point in the training data set should contain two parts: flow features and a label (active or inactive). The features

TABLE I
FEATURE LIST

Feature	Description
1_{tcp}	1 if the flow is a TCP flow, 0 otherwise
t_{idle}	the time since the last packet arrival of a flow
t_{ia}, t_{is}	mean, std. dev. of the inter-arrival times of the last N_{pkt} packets of a flow
$l_i (0 \leq i < N_{\text{pkt}})$	the lengths of the last N_{pkt} packets of a flow

are used to characterize the state of the flow from which the data point was generated. In this study, we use the features listed in Table I. t_{idle} captures the time since the last packet was seen from the flow, and a larger t_{idle} generally means the flow entry is more likely to be inactive. t_{ia} and t_{is} characterize the packet inter-arrival time distribution of the flow. We also capture the lengths of the last N_{pkt} packets in the l_i features. Thus, the feature vector has the form $v = (1_{\text{tcp}}, t_{\text{idle}}, t_{\text{ia}}, t_{\text{is}}, l_1, l_2, \cdots, l_{N_{\text{pkt}}}) \in \mathbb{R}^{N_{\text{pkt}}+4}$, for each data point.

Algorithm 1 Dataset Generation

Require: Packet trace, $t_{\max}$, $N_{\max}$, t_{interval}, N_{pkt}, $t_{\text{threshold}}$
1: **for** TCP/UDP packet p whose arrival time $t_p \leq t_{\max}$ **do**
2: If the next packet belonging to the flow containing p will arrive after $t_{\text{threshold}}$ or p is the last packet, label the flow as inactive. Otherwise, label it as active.
3: **if** p cannot match any flow entry **then**
4: **if** the size of the flow table is $N_{\max}$ **then**
5: Output the features and label of each flow entry which is recorded $\geq t_{\text{interval}}$ ago or updated;
6: Set each flow entry as non-updated and t_{record} of each flow entry as t_p;
7: Evict a random flow entry;
8: **end if**
9: Insert the flow entry subject to p in the flow table.
10: Set the flow entry as updated.
11: **else**
12: Update the features of the flow entry referred by p
13: **end if**
14: **end for**

With these features, we can generate a training dataset from real network packet traces. The generation process is described in Algorithm 1, which simulates the arrival of packets and updates the feature vector of the corresponding flow entry when a packet arrives. The algorithm also simulates flow entry installation and eviction. As packets arrive, flow entries are installed in the flow table. Whenever the flow table reaches its capacity, the features and label of every flow entry are output as a data sample (see line 5). The algorithm labels a flow entry as inactive (positive) when there is no packet from the flow in the next $t_{\text{threshold}}$ length of time in the trace. This is reasonable if $t_{\text{threshold}}$ is long enough (e.g., 1 hour).

Note that all identified features are time-varying except for 1_{tcp}. In particular, t_{idle} for a flow keeps changing even if there are no packet arrivals for the flow. Thus, there could be thousands of data samples which are exactly the same except that their t_{idle} values are slightly different. This massive redundancy of data samples would not only provide no extra information to the ML model but it could also introduce bias in the trained model due to the frequency of redundant samples in the data set. This would also increase the computational overhead of training. To prevent these negative effects, we employ a variable, t_{record}, to record the last time that the features and label of a flow entry were output as a data sample and we do not allow a new data sample to be output for that flow until at least time $t_{\text{record}} + t_{\text{interval}}$ (see line 5).

Another issue is which policy (e.g., random, LRU, FIFO) to use for flow entry eviction when the flow table overflows during training dataset generation. In machine learning, we would like the training data and test data to come from the same underlying distribution so that the trained model can achieve low generalization error. The LRU and FIFO policies both have some preference about which flow entry should be evicted when the flow table overflows. These preferences are clearly different from the learned ML policy, which would mean that a training dataset generated with LRU or FIFO policy would have a very different distribution from the testing one (used by the learned ML policy). However, the random eviction policy has no preference about which flow entry should be evicted, and thus the distribution of the training dataset generated by the random policy will be closer to that of the learned policy. Therefore, to generate the training dataset, we employ the random eviction policy (see line 7).

B. Offline Model Training: Model Tuning

With the collected dataset, we need to select an appropriate machine learning algorithm and tune its hyperparameters to achieve the best performance. Many algorithms can be used for classification problems, such as nearest neighbor, support vector machine, decision tree, random forest, and multiple layer perception [24]. To select the best machine learning algorithm for STEREOS and tune its hyperparameters, we need to select an appropriate performance metric. There are many performance metrics for classification, such as classification accuracy, recall, and F1-score [24]. For flow entry eviction, on one hand, we want to minimize false positives (i.e., active flows misclassified as inactive) to minimize wrong evictions. On the other hand, if an inactive flow entry is misclassified as active, then it might never be evicted, which wastes precious flow table resources. Thus, false negatives should also be minimized. Based on these observations, we use F1 score as the performance metric because a high F1 score indicates that both false negatives and false positives are low.

Based on F1 score, we use a K-fold rolling-origin cross validation, as shown in Fig. 2, to evaluate the performance of different machine learning models with different hyperparameter configurations [25]. K-fold rolling-origin cross validation is a common approach to fine-tune model hyperparameters for time series data, where the whole time is first split into a training range and a testing range such that all observations in the training range occurred prior to any observation in the testing range. The training range is further divided into 2K roughly equal time slices ($K = 3$ in Fig. 2). For fold

$k = 1, 2, \cdots, K$, we fit a machine learning model with its hyperparameters to the first $K+k-1$ time slices, and compute its F1 score in classifying the $(K+k)$th part (a.k.a., validation slice). Then we get the average F1 score of these K fold cross validations. For every machine learning model, we do this for many hyperparameter values and choose the values that maximize the average F1 score.

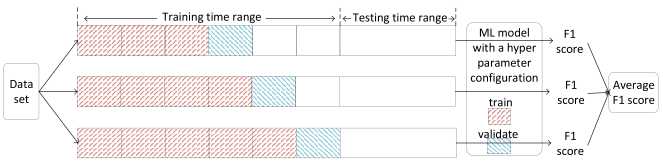

Fig. 2. K-fold rolling-origin cross validation (K = 3) for tuning the hyperparameters of classification models.

C. Online Flow Table Eviction

Once offline training is finished, we apply the trained binary classification model to online flow table eviction. The trained model not only predicts whether one flow entry is inactive but also gives the confidence of the prediction, i.e., the probability that the flow entry is inactive. We rely on these probabilities for online flow entry eviction, as shown in Algorithm 2. For a binary classification model h trained from the collected dataset, $h(v_e)$ gives the inactive probability for flow entry e with feature vector v_e, i.e., P^e_{inactive}. When the flow table reaches capacity, we apply the model h to calculate P_{inactive} for the flow entries in the table and we use those probabilities to choose a flow entry for eviction as discussed below.

Algorithm 2 Online Flow Entry Eviction

Require: Trained model h, $P_{\min}$, t_{interval}
1: **while** a packet p is arriving at the switch **do**
2: **if** p is matched with a flow entry e_p **then**
3: update the feature vector v_{e_p} associate with e_p
4: **else**
5: **if** the flow table is overflow **then**
6: $isEvicted \leftarrow false$
7: **for** every flow entry e in the flow table **do**
8: **if** v_e is updated or P^e_{inactive} is updated t_{interval} ago **then**
9: $P^e_{\text{inactive}} \leftarrow h(v_e)$
10: **if** $P^e_{\text{inactive}} > 0.9$ **then**
11: 1) evict the entry e
12: 2) $isEvicted \leftarrow true$
13: **break**
14: **end if**
15: **end if**
16: **end for**
17: **if** not $isEvicted$ **then**
18: $e^* = argmax\{P^e_{\text{inactive}}\}$
19: **if** $P^{e^*}_{\text{inactive}} > P_{\min}$ **then**
20: evict the flow entry e^*
21: **else**
22: evict the least recently used flow entry
23: **end if**
24: **end if**
25: **end if**
26: send flow setup request to the controller to install flow entry for packet p
27: **end if**
28: **end while**

Calculating the inactive probabilities for all entries when the flow table overflows is computationally intensive and would place a heavy burden on the limited CPU of the SDN switch. Therefore, we set a threshold (0.9 in the experiments reported in Section IV) on P_{inactive}, so that as soon as we find an entry with P_{inactive} exceeding the threshold, we choose it for eviction immediately without calculating the rest of the P_{inactive} values.

In addition, it is inefficient to do classification on the same flow entry again if its feature vector has only a small change. For example, we do not want to classify a flow entry again if its feature vector stays the same except that t_{idle} is 1 millisecond different. Therefore, similar to how we saved computation time in the training phase, if there are no new packets for a flow within a time t_{interval}, we do not reclassify it during that time. This further reduces the computation burden at flow table overflow time so that, in most cases, only flows that had packets arriving at the switch since the last table overflow will need to be reclassified.

The last issue is how to handle inactive flow entries that are misclassified as active. For inactive flow entries, only t_{idle} will change as time elapses and it is possible that these entries will remain misclassified for a long time. To make matters worse, these inactive flow entries could accumulate over time and occupy most of the flow table. To address this, if the flow entry with largest inactive probability does not satisfy $P^{e*}_{\text{inactive}} \geq P_{\min}$, we evict the LRU flow entry. Otherwise, the flow entry with maximum P_{inactive} will be evicted (unless one with $P_{\text{inactive}} > 0.9$ was already evicted) (see line 20). In this way, misclassified inactive flow entries, whose t_{idle} tends to be large, can be removed if there is no other flow with a high inactive probability. Choice of $P_{\min}$ is discussed in Section IV.

D. Overhead of STEREOS

STEREOS requires storing feature values for every flow entry in the flow table. The percentage overhead this incurs depends on the storage requirement for these features and the size of each flow entry. Each flow entry contains match fields, priority, counters, instructions, timeouts, cookie, and flags. The priority, timeout, cookie, and flag fields together require 24 bytes. The match fields are described using OpenFlow Extensible Match (OXM) format, and each OXM value requires between 5 and 259 bytes. For the typical 5-tuple flow match fields, 4 OXMs are required and 28 bytes are consumed. Depending on which counters are included, the counter fields consume from 4 to 24 bytes. The number of bytes occupied by instructions depends on what actions are included. For example, if one output action is included, then 24 bytes are required. In summary, the storage requirement for one flow

TABLE II
SUMMARY OF PACKET TRACES USED FOR CASE STUDY

Packet trace	Duration (s)	Number of packets	Number of flows	TCP flow percentage
UNIBS0930	81,203	4,189,545	43,489	97.3
UNIBS1001	32,407	3,321,426	39,730	97.4
UNIV1	3,914	17,131,142	439,133	67.4
UNIV2	3,558	35,243,160	38,016	8.70

entry varies but the classical simple 5-tuple output flow entry requires 80 bytes. To achieve more precise control (which is a common case in OpenFlow networks), more complicated entries are necessary, which will require more bytes per entry.

The feature vector for STEREOS includes idle time, and the packet lengths and inter-arrival times of the last N_{pkt} packets. If the time features are quantized with B_t bytes and packet lengths are quantized with B_l bytes, then we require $(B_t + B_l)N_{pkt}$ bytes to store the feature vector for each flow entry. In Section IV, we discuss appropriate values of N_{pkt}, B_t, and B_l and we use those to quantify the overhead of STEREOS.

IV. CASE STUDIES

In this section, we present case studies based on four real packet traces collected from university data centers. UNIBS0930 and UNIBS1001 were collected on an edge router of the campus network of the University of Brescia on two working days, namely 09/30/2009 and 10/01/2009 [26]. They are composed of traffic generated by a set of twenty workstations running the ground truth client daemon. UNIV1 and UNIV2 were collected from two university data centers by the authors of [5]. For simplicity, we only consider TCP and UDP flows in the case studies. We summarize these four packet traces in Table II.

A. Dataset Collection

We follow Algorithm 1 to collect datasets for the four packet traces with 1K, 2K, and 4K flow tables ($N_{\max}$). The very latest high-end commercial OpenFlow switches are equipped with larger flow tables (e.g., 16K) to handle traffic demands in today's enormous data centers. However, the traces used here are collected from campus networks around 10 years ago. Therefore, we consider 1K, 2K, and 4K flow tables which reflect hardware capacity from the same time frame and are consistent with other studies such as [27]. The results shown herein should generalize to larger networks with commensurately larger numbers of flows and flow table sizes.

The generated datasets are summarized in Table III. Training flows are flows that start in the training interval, $(0, t_{max})$, of the packet trace, non-cross flows are flows that start after t_{max}, and cross flows are flows that start before t_{max} and end after t_{max}. We distinguish cross flows and non-cross flows so that we can analyze whether the patterns learned from training flows can be applied to flows that were not in the training dataset, i.e. the non-cross flows. This is significant because the benefits of STEREOS would degrade as time elapsed if learned patterns did not apply to new (non-cross) flows.

TABLE III
SUMMARY OF GENERATED DATASETS

Packet trace	t_{max} (s)	t_{int} (s)	Training flows	Non cross flows	Cross flows	Table size	Dataset size
UNIBS 0930	10000	20	11283	33076	870	1K	563537
						2K	829985
						4K	1037730
UNIBS 1001	5000	10	7284	33181	735	1K	23097
						2K	381279
						4K	438524
UNIV1	600	1	95503	398395	54765	1K	874663
						2K	1420374
						4K	2472854
UNIV2	550	1	19073	34540	15597	1K	873753
						2K	1336900
						4K	2229184

TABLE IV
HYPERPARAMETER SEARCH SPACE FOR MODEL TUNING

Algorithm	Hyperparameter	Search space
Random Forest (RF)	n_estimators	[10, 20, 30]
	criterion	[entropy, gini]
	max_depth	[10, 20, 30]
Decision Tree (DT)	criterion	[entropy, gini]
	max_depth	[10, 20, 30]
Ada Boosting	n_estimators	[10, 20, 30]
	learning_rate	[0.8, 0.9, 1.1]
Gradient Boosting Tree (GBT)	n_estimators	[10, 20, 30]
	subsample	[0.6, 0.8, 1.0]
	learning_rate	[0.01, 0.1, 0.5]
	max_depth	[10, 20, 30]
Linear Regr. (LR)	penalty	[L1, L2]
	C	[0.01, 0.1, 1.0]
Neural Net (NN)	alpha	[0.01, 0.1, 1.0, 10.0]
	learning_rate	[constant, invscaling, adapt.]
	learning_rate_init	[0.01, 0.1, 1.0, 10.0]
	hidden_layer_sizes	[(30), (15, 15), (10, 10, 10)]

B. Offline Training Results

We use `scikit-learn,pedregosa2011scikit`, an open source machine learning library in Python, for selecting the best machine learning models. This library provides many classification algorithms, as well as the APIs for model selection based on cross-validation. As shown in Fig. 2, each piece of the last five periods $t_{max}/10$ is a validation set and the previous time's data is the training set. In this way, we can apply 5-fold rolling-origin-cross-validation to tune the hyperparameters of different machine learning algorithms.

We considered the seven classification algorithms shown in Table IV. We did not consider the K nearest neighbor algorithm because it requires storage of the whole dataset which is not feasible for OpenFlow switches with limited memory. We also did not consider the support vector machine (SVM) algorithm because of its extremely high computational overhead for training. Each of the considered algorithms has many hyperparameters and we only considered some of the most important ones, which are shown in Table IV. For all of the hyperparameters not specified in Table IV, we used the default values provided by the library.

The F1 scores for all possible combinations of the hyperparameter values specified in Table IV were evaluated by 5-fold rolling-origin-cross-validation. From this evaluation,

TABLE V
MODEL SELECTION RESULTS

Packet trace	Table size	Best model in terms of F1 score	Training	Validation on cross flows	Validation on non-cross flows	Validation
UNIBS 0930	1K	GBT { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.98784	0.98380	0.93758	0.96171
	2K	GBT { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.99499	0.99303	0.95014	0.98106
	4K	GBT { n_estimators:30; subsample:0.6; learning_rate:0.1; max_depth:10; }	0.99775	0.99494	0.94634	0.98811
UNIBS 1001	1K	RF{ n_estimators:30; criterion:entropy; max_depth:20; }	0.99662	0.99454	0.93710	0.97660
	2K	GBT { n_estimators:30; subsample:1.0; learning_rate:0.1; max_depth:10; }	0.99497	0.99661	0.94930	0.98861
	4K	RF{ n_estimators:30; criterion:gini; max_depth:20; }	0.99928	0.99901	0.94492	0.99526
UNIV1	1K	RF{ n_estimators:30; criterion:entropy; max_depth:20; }	0.93629	0.79215	0.93148	0.90258
	2K	RF{ n_estimators:30; criterion:entropy; max_depth:20; }	0.96081	0.91783	0.95195	0.93946
	4K	RF{ n_estimators:30; criterion:entropy; max_depth:20; }	0.97890	0.96687	0.95983	0.96435
UNIV2	1K	GBT { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.94876	0.69402	0.93979	0.89006
	2K	GBT { n_estimators:30; subsample:0.6; learning_rate:0.1; max_depth:20; }	0.96857	0.84351	0.94421	0.90926
	4K	GBT { n_estimators:20; subsample:0.6; learning_rate:0.1; max_depth:20; }	0.98324	0.94429	0.95206	0.94790

we picked the best algorithm and its best hyperparameters, and the results are shown in Table V. As can be seen in the table, either GBT or RF is the best algorithm for all four packet traces (in fact, their achieved F1 scores fall within about 0.5% of each other in all cases). These two methods are both ensemble methods, which are unlikely to overfit.

We also employed the concepts of cross and non-cross flows during the cross-validation to validate the ML model's accuracy on non-cross flows in terms of F1 scores. For the kth fold validation, cross flows are flows that start within the first $K + k - 1$ time slices and end within the $(K + k)$th slice, while non-cross flows start and end within the $(K + k)$th slice. In this way, we can get F1 scores for both cross and non-cross flows for each fold validation, which are shown in Table V. Note that the F1 scores for the non-cross flows are all quite high and, for the UNIV packet traces, they are actually higher than the F1 scores for the cross flows, while they are only slightly smaller than those of the cross flows in the UNIBS traces. These results show that the trained model can be well generalized to classify unseen flows. If this was not true, the F1 scores for the non-cross flows would be much lower than those of the cross flows since the model does not see non-cross flows during training. This means that there are some common patterns that inactive flows exhibit. The model learns these patterns from the training flows and can then identify whether an unseen flow is active or not. This is extremely important since the model is most likely to be applied to non-cross flows if it is used in real systems for a long time. Otherwise, it would be impractical to apply the learned model to do the classification since the model would soon be outdated.

After we select the best model with its best hyperparameters for each dataset, we train that model over the whole training duration $(0, t_{max})$, save the trained model, and then use the model for online flow entry eviction.

C. Model Size Trade-Off

When we did model selection as described in the last subsection, we only considered the F1 score of each model. However, the F1 scores do not provide a complete picture of model performance. First, the F1 score does not perfectly describe the performance of a model when it is applied to flow entry eviction. The primary goal is to minimize incorrect flow entry evictions. While the F1 score makes a trade-off between false positives and false negatives, which are both related to incorrect flow entry evictions, it may not perfectly reflect the models' relative performances in reducing the number of capacity misses. Second, because OpenFlow switches have limited memory, if a model achieves a slightly smaller F1 score but occupies a much smaller memory footprint, it may be a better choice for implementation in practice.

In this section, we study trade-offs between performance and memory size of the models. Here, we directly study the number of capacity misses, which is collected from online eviction simulations, to evaluate the performance of a model. Capacity miss is a concept borrowed from computer architecture, and it refers to a flow entry miss (i.e., no flow entry in the flow table can be matched with the incoming packet) that is due to the limited table size. The number of capacity misses is equal to the number of evictions of active flows, because evicting an active flow entry will later result in a flow entry re-installation, which is a capacity miss. To collect the number of capacity misses, we build a simulator which is similar to the one used in dataset generation except that the flow entry eviction policy can be chosen from LRU, STEREOS, and ALFE [18]. The packets in the trace are replayed to the simulator and flow installations and evictions are simulated to count the number of capacity misses.

Table VI shows the performance of different models with their sizes, where the ratio of capacity misses of STEREOS to LRU (i.e., β) is used to compare STEREOS and LRU. Note we do not show the results of UNIV2 because it is not the typical case in real networks as will be discussed in Section IV-G. From the table, we make the following observations:

1) The model with the highest F1 score does not always achieve the best performance in terms of reducing capacity misses. For example, in the case of UNIBS0930 with 4K flow table, the best model in terms of F1 score results in *more* capacity misses than LRU. However, the RF model (`n_estimators`: 20, `criterion`: entropy, `max_depth`: 20) produces only 47.2% of the capacity misses of LRU, although its F1 score is slightly smaller than the "best" model. This phenomenon may be due

TABLE VI
PERFORMANCE AND MODEL SIZE TRADE-OFF, β IS THE RATIO OF CAPACITY MISSES OF ML POLICY TO LRU

Packet trace	Table size	F1 score	Model size (MB)	Model hyperparameters	β
UNIBS 0930	1K	0.96060	1.79	**GBT** { n_estimators:20; subsample:0.6; learning_rate:0.1; max_depth:10; }	0.354
		0.95104	0.08	DT{ criterion:entropy; max_depth:10; }	0.685
		0.96171	2.75	GBT* { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.383
	2K	0.98037	1.70	**GBT** { n_estimators:20; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.344
		0.97606	0.08	DT{ criterion:gini; max_depth:10; }	0.882
		0.98106	2.65	GBT* { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.397
	4K	0.98759	1.84	GBT { n_estimators:20; subsample:0.6; learning_rate:0.1; max_depth:10; }	1.078
		0.98531	13.48	**RF** {n_estimators: 20, criterion: entropy, max_depth: 20}	0.472
		0.98811	2.73	GBT* { n_estimators:30; subsample:0.6; learning_rate:0.1; max_depth:10; }	1.104
UNIBS 1001	1K	0.97606	8.01	RF { n_estimators:20; criterion:entropy; max_depth:20; }	0.249
		0.97502	1.02	**GBT** { n_estimators:20; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.241
		0.97645	13.08	RF* { n_estimators:30; criterion:entropy; max_depth:20; }	0.222
	2K	0.98833	0.98	**GBT** { n_estimators:20; subsample:0.6; learning_rate:0.1; max_depth:10; }	0.492
		0.98604	0.05	DT{ criterion:gini; max_depth:10; }	0.944
		0.98857	1.50	GBT* { n_estimators:30; subsample:1.0; learning_rate:0.1; max_depth:10; }	0.597
	4K	0.99515	5.21	**RF** { n_estimators:20; criterion:entropy; max_depth:20; }	0.508
		0.99480	0.93	GBT { n_estimators:20; subsample:0.8; learning_rate:0.1; max_depth:10; }	1.773
		0.99526	8.24	RF* { n_estimators:30; criterion:gini; max_depth:20; }	0.514
UNIV1	1K	0.89705	3.31	**GBT** { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.492
		0.89364	2.22	GBT { n_estimators:20; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.498
		0.90258	41.07	RF* { n_estimators:30; criterion:entropy; max_depth:20; }	0.501
	2K	0.93673	14.42	RF { n_estimators:10; criterion:entropy; max_depth:20; }	0.606
		0.93385	3.34	**GBT** { n_estimators:30; subsample:0.6; learning_rate:0.1; max_depth:10; }	0.590
		0.93946	46.47	RF* { n_estimators:30; criterion:entropy; max_depth:20; }	0.545
	4K	0.96389	30.88	RF { n_estimators:20; criterion:entropy; max_depth:20; }	0.929
		0.95862	3.29	**GBT** { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.853
		0.96435	46.14	RF* { n_estimators:30; criterion:entropy; max_depth:20; }	0.932

Note: The best model in terms of F1 score is marked with "*", and the selected model considering both model size and performance is marked in bold.

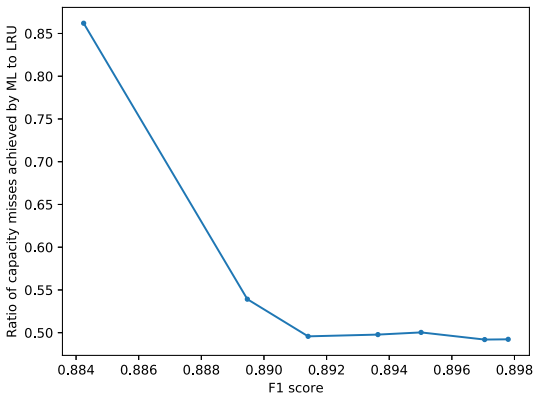

Fig. 3. The effects of F1 score on STEREOS.

to the fact that there is some randomness in terms of the number of capacity misses for models with close F1 score, which is shown in Fig. 3. This figure was produced by running simulations on UNIV1 trace with 1K flow table, where the model used for evicting flow entries is GBT (subsample:0.8; learning_rate:0.1; max_depth:10) with `n_estimators={5, 10, 15, 20, 25, 30, 35}`. The figure shows that, as the F1 score varies within the range of [0.892, 0.898], the number of capacity misses fluctuates little but within the range of [0.884, 0.892] there is a very large variation.

2) Smaller models can some times achieve fewer capacity misses. For example, for the UNIV1 trace with 1K flow table, the "best" model (i.e., RF model) has a size of 41.07 MB and achieves 50.1% capacity misses of the LRU policy. However, the GBT model (`n_estimators`:30; `subsample`:0.8; `learning_rate`:0.1; `max_depth`:10) only requires 3.31 MB and achieves very close to the miss rate of the RF model (49.2% misses).

Based on these observations, we re-select the models for the different traces considering both model size and performance, and show the best performing ones in bold in Table VI.

D. Tuning P_{min}

According to Algorithm 2, $P_{\min}$ is an important parameter for STEREOS which controls how much STEREOS should depend on the LRU policy. Therefore, it is significant to tune P_{min} for implementing STEREOS. On one hand, small $P_{\min}$ allows the switch to evict flow entries which are classified as inactive with low confidence. In this case, it is highly possible that a misclassified active flow entry will be evicted. On the other hand, large $P_{\min}$ would prevent the switch from evicting inactive flow entries which are not identified by the trained model with very high confidence. Actually, with large $P_{\min}$, the switch would heavily rely on the LRU policy (line 22 in Algorithm 2) for eviction instead of the machine learning one. Therefore, we need to tune P_{min} such that as many inactive flow entries as possible satisfy $P_{\text{inactive}} > P_{\min}$ and as few active entries as possible do not meet this.

To find the appropriate $P_{\min}$, we collected the number of capacity misses achieved by STEREOS with different $P_{\min}$ on the training periods through online eviction simulations, which are shown in Fig. 4. As we can see, the number of capacity misses decreases up to a certain $P_{\min}$, then increases

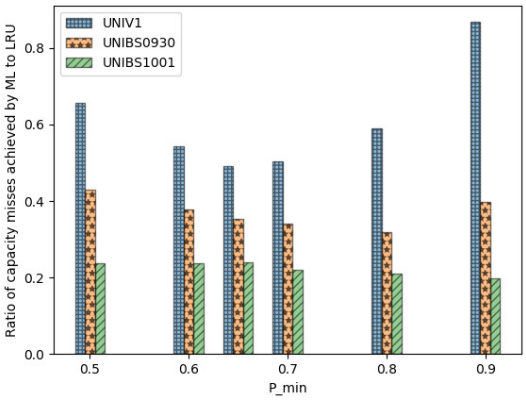

Fig. 4. The effects of P_{min} on STEREOS with 1K flow table.

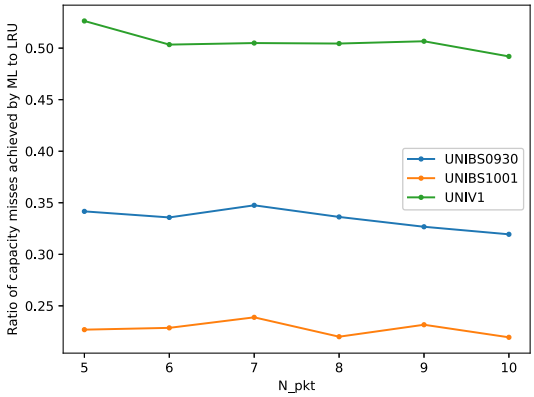

Fig. 5. The effects of N_{pkt} on STEREOS with 1K flow table.

as $P_{\min}$ grows. For example, the number of capacity misses is reduced by 24% when $P_{\min}$ is changed from 0.5 to 0.65, and it is increased by 74% from 0.65 to 0.9. In addition, the optimal $P_{\min}$ values that achieve the minimum numbers of capacity misses are different for the three considered packet traces. The optimal values are 0.65 for UNIV1, 0.8 for UNIBS0930, and 0.9 for UNIBS1001. This is because the ML models used by STEREOS achieve different classification accuracies on these traces, as shown in Table V. In general, $P_{\min}$ is larger if the model achieves higher accuracy. We used these tuned $P_{\min}$ values in the results reported in the rest of this section.

E. Tuning N_{pkt}

As we discussed in Section III-D, the overhead of STEREOS is directly related to N_{pkt}. In the above experiments, we set N_{pkt} to be 10, which may incur high overhead. In this subsection, we study how N_{pkt} can affect the performance of STEREOS. We first generate the datasets with $N_{\text{pkt}} = 5, 6, 7, 8$, and 9 for UNIV1 and UNIBS traces. The selected models in Table VI are then trained on these datasets and used in online flow entry eviction simulations.

The simulation results are shown in Fig. 5. As we can see, the number of capacity misses tends to increase as N_{pkt} decreases for UNIV1. For example, the number of capacity misses is increased by 7%, 16%, and 17.6% when N_{pkt} is reduced from 10 to 5 for the 1K, 2K, and 4K table, respectively. This is because a larger N_{pkt} can, in general, provide more information for the model and thus help increase the classification accuracy. On the other hand, larger N_{pkt} means the OpenFlow switches would need more memory to store the feature vectors. For example, when N_{pkt} is increased from 5 to 10, the memory cost of storing feature vectors almost doubles. However, for UNIBS0930 and UNIBS1001 traces, the effect of N_{pkt} on STEREOS is not significant and even shows some degree of fluctuation. This is because UNIBS traces contain fewer flows than UNIV1, and the ML models with $N_{\text{pkt}} = 5$ can achieve very high F1 scores (around 0.99). In this case, larger N_{pkt} only contributes a small amount to the models' accuracy. In summary, $N_{\text{pkt}} = 5$ is good for all three traces to make a trade-off between memory consumption and classification accuracy in practical implementation and so we use that value in the remainder of the paper.

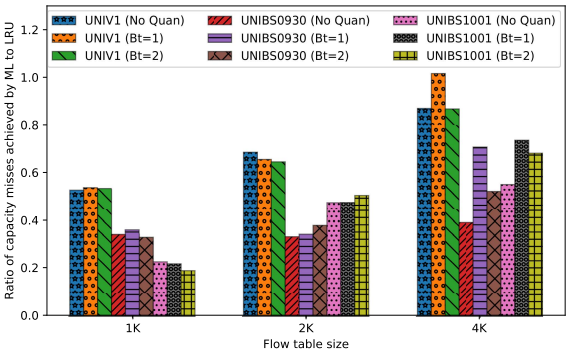

Fig. 6. The effects of feature quantization on STEREOS.

F. Feature Quantization and STEREOS Overhead

The overhead of STEREOS depends not only on N_{pkt} but also on the quantization levels B_t and B_l. We use uniform quantization, which removes values above v_{max} and uses the same quantization level for all values below v_{max}.

The size of a packet is limited. For example, Microsoft Windows computers default to a maximum packet size of 1500 bytes for broadband connections, and the maximum transmission unit (MTU) of Ethernet is 1500 bytes. Thus, we quantize packet length with one byte, and we set v_{max} for packet length to be 1500.

We tested both 1 byte and 2 bytes for quantizing the time features (t_{idle} and t_{ia}), and the results are shown in Figure 6. The figure shows that, in most cases, the performance of STEREOS slightly degrades when quantization is applied. For example, for UNIV1 with a 1K flow table, the number of capacity misses is increased by 2% and 1.4% with 1 byte and 2 bytes quantization, respectively. In other cases, quantization actually slightly improves the performance of STEREOS. The reason for this might be that quantization can drop some unnecessary information for training, and thus increase the model's accuracy. Overall, Figure 6 shows that 1 byte for quantizing the time features achieves close performance to the case where no quantization is applied.

We conclude that the overhead of STEREOS is acceptable since with the values chosen for N_{pkt}, B_t, and B_l, as described above, only 10 extra bytes are required for each flow entry. As discussed in Section III-D, basic storage information for a flow entry uses a minimum of 80 bytes.

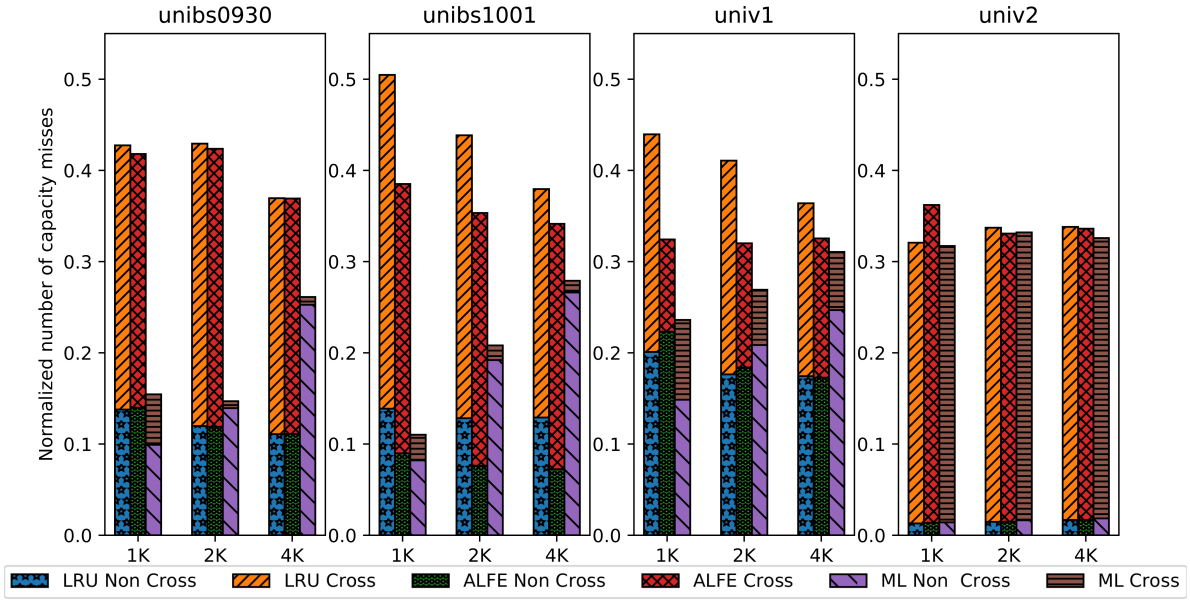

Fig. 7. Number of capacity misses for STEREOS, LRU, and ALFE policies.

TABLE VII
MAIN PARAMETERS FOR ONLINE EVICTION SIMULATIONS

Model	Models marked in bold in Table VI
N_{pkt}	5
$P_{\min}$	0.65 for UNIV traces, 0.8 for UNIBS0930, 0.9 for UNIBS1001
Pkt feature quan	1 byte, v_{max}=1500
Time feature quan	1 byte, v_{max}=250 for UNIBS traces, and v_{max}=10 for UNIV traces
t_{interval}	Same as Table III

G. Online Simulation Results

So far, we have selected ML models for each trace with different flow table sizes, tuned their hyperparameters, N_{pkt} and $P_{\min}$, and chosen the feature quantization levels. Combining all of these components, we performed online flow entry eviction simulations on the testing periods of all four traces with 1K, 2K, and 4K flow tables. We performed detailed simulations of flow installation and flow eviction for the STEREOS eviction policy along with LRU and ALFE [18] policies for comparison, using a custom Python simulator. The Python simulations allowed us to calculate and compare miss rates for the different policies. Using the miss rates for STEREOS and LRU found through these simulations, we also performed ns-3 simulations to evaluate overall performance in a more realistic full network setting. All simulations were done on a 3.6 GHz Intel Xeon processor with 8 MB of RAM. The main simulation parameters used in the Python simulations are summarized in Table VII.

Fig. 7 shows the performance of STEREOS in terms of the number of capacity misses, where the number of capacity misses of one packet trace for each policy is normalized with respect to the total number of capacity misses across ML, LRU, and ALFE policies for that trace. From Fig. 7, we have the following observations:

1) *For the two UNIBS traces and the UNIV1 trace, the performance gain of STEREOS over LRU is substantial, especially for 1K and 2K flow table sizes.* For those cases, STEREOS' improvement ranges from 45% to 78% on the three traces, compared with the LRU policy. Note that reducing capacity misses is extremely important for OpenFlow network performance. Reducing capacity misses lowers the number of `PacketIn` events and thus relieves the load on control channels and controllers, and it also means that fewer TCP transmissions are interrupted. In this respect, the over 45% performance gain in terms of capacity misses achieved by STEREOS is extremely significant. As flow table size increases, the benefits of STEREOS are reduced somewhat. With 4K flow table size, for example, STEREOS reduces the number of capacity misses on the UNIV1 trace by 15% and on the UNIBS0930 trace by 30%, compared with LRU. However, even those gains are significant given the critical importance of minimizing capacity misses.
We note that, for these traces, a 4K flow table size is large enough to hold almost all of the active flows, which is why the performance of STEREOS, relative to LRU, drops for this case. While flow table capacities in the newest SDN switches are even greater than 4K, current software-defined data center networks are much larger than the networks used to collect these traces and they will, therefore, have far more active flows than in our simulations. In fact, physical limitations on flow table implementation prevent flow table size growth from keeping up with network size and traffic growth. Thus, capacity misses are likely to be an even greater issue in the future, meaning that the benefits of STEREOS should be even more significant moving forward.
2) *The number of capacity misses on cross flows for LRU is extremely high, whereas those misses are reduced to a very low level for the ML policy in most cases.* As observed earlier, the packet inter-arrival times of

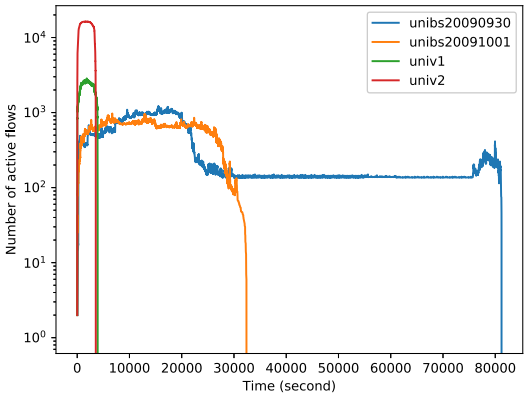

Fig. 8. The number of active flows.

cross flows are larger than those of non-cross flows and flow entries with larger packet inter-arrival times are more likely to be evicted by LRU. On the surface, from looking at Fig. 7, it appears that the ML policy performs *worse* than LRU on non-cross flows, which seemingly contradicts our conclusion that STEREOS can learn patterns of non-cross flows even though they don't appear in the training data set. However, the reason for this seeming anomaly is simply that LRU is so poor at classifying cross flows that an overwhelming number of its capacity misses result from those flows, not leaving many opportunities for non-cross flows to be wrongly evicted. Since ML does so much better on cross flows as shown in Fig. 9, the capacity misses that it does incur happen more frequently with the non-cross flows. However, the overall rate of capacity misses is reduced substantially in almost all cases with the ML policy.

3) *The performance of STEREOS is lower on the UNIV2 trace; however, this trace is not representative of typical network traffic scenarios.* For the UNIV2 trace, STEREOS achieves less than 5% performance gain over LRU. This is because the number of active flows in UNIV2 is always much larger than the flow table size, as shown in Fig. 8. This means that, at any given time, most of the entries in the flow table are active, and thus STEREOS cannot outperform LRU since STEREOS is based on the assumption that some of the flow entries are inactive. Fortunately, this situation is not typical in real networks – it occurs only because most of the flows in UNIV2 are UDP flows. As shown in Table II, more than 90% of the flows in the UNIV2 traces are UDP flows, while in typical data center traffic mixes, UDP data volume is only 3% $\sim$ 9% of TCP data [19]. We note that STEREOS performs well with UDP percentages well above these typical values since, for the UNIV1 traces, about 33% of the flows are UDP flows and STEREOS substantially outperforms LRU on those traces.
4) Compared to another eviction policy from the literature, ALFE [18], the performance improvement for STEREOS is as large as it is compared to LRU in some cases (see UNIBS0930 traces). However, for the UNIBS1001 and UNIV1 traces, ALFE outperforms LRU but is worse than STEREOS. The improvement of STEREOS compared to ALFE on those traces ranges from 5% to 72%, with an average improvement of 30%. ALFE's performance is heavily dependent on existence of a heavy tail in the packets per flow distribution, i.e. a situation where a few "elephant" flows produce a disproportionally large percentage of packets. In the absence of elephant flows, ALFE's performance degrades to that of LRU (or even slightly worse) [18]. The presence or absence of elephant flows is, therefore, a likely explanation for the variation in ALFE's performance across the different traces. *Because STEREOS is trained to learn the specific characteristics of a particular network environment, its performance improvement remains high across a range of types of traffic.*

We also investigate the number of active flow entries in the flow table. Fig. 10 shows the active flow entries in the flow table with machine learning and LRU policies on the UNIV1 packet trace. As can be seen, the number of active flow entries in the flow table with STEREOS is much larger than LRU. On average, STEREOS can increase the usage of the flow table with 1K, 2K, and 4K capacity by 58%, 60%, and 54% respectively, compared with LRU. This significant improvement is achieved because STEREOS can correctly identify and evict inactive flow entries when flow table overflow occurs. By contrast, LRU may frequently remove active flow entries and leave inactive flow entries in the flow table. From Fig. 7 and Fig. 10, we can conclude that our machine learning based flow entry eviction policy can achieve significant performance gains compared with LRU policy.

Finally, we investigated how much performance gain STEREOS can achieve in terms of network-wide metrics such as throughput, delay, and packet loss rate through simulations in an enhanced version of the packet-level simulator ns-3 we developed to evaluate different flow entry eviction methods. Due to space limitations, we do not present the details of the simulations in this paper, but they can be found in [29]. The simulation results show that, compared with the LRU eviction, STEREOS can reduce control overhead (i.e., the number of OpenFlow messages exchanged between the controller and switches) by 30%, increase the throughput by 19%, and reduce the packet loss rate by 70%. This is because STEREOS can mitigate wrong flow entry evictions, meaning that fewer expensive flow setups are required. This leaves more existing connections undisturbed, maintains free space in the SDN buffers (e.g., the datapath buffer that stores unmatched packets), and keeps the packet loss rate low.

H. Model Interpretation

So far, we have done a detailed investigation demonstrating that STEREOS outperforms LRU. In this section, we attempt to understand why STEREOS performs better. This relates to how to interpret a machine learning model, which is a problem that is still far from being solved [30].

Since this case study involves tens of models, we only try to interpret the final model for UNIV1 trace with 1K flow table, which is GBT (`n_estimators`:30; `subsample`:0.8;

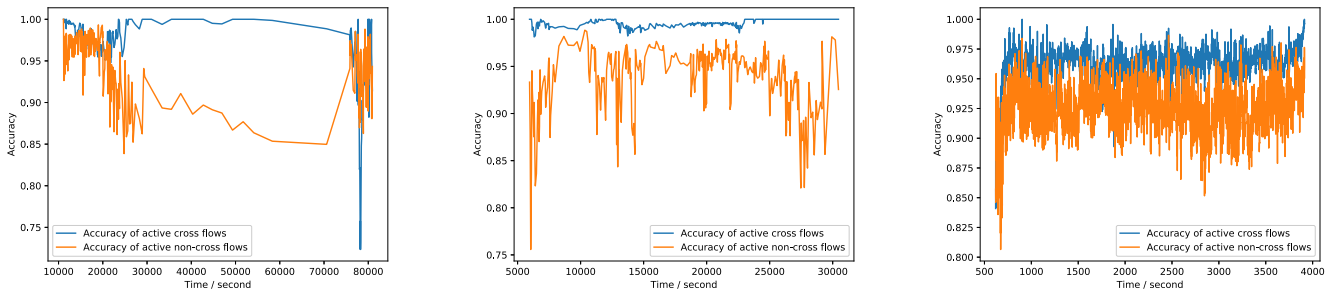

Fig. 9. Classification accuracy with 2K flow table: (a) UNIBS0930; (b) UNIBS1001; (c) UNIV1.

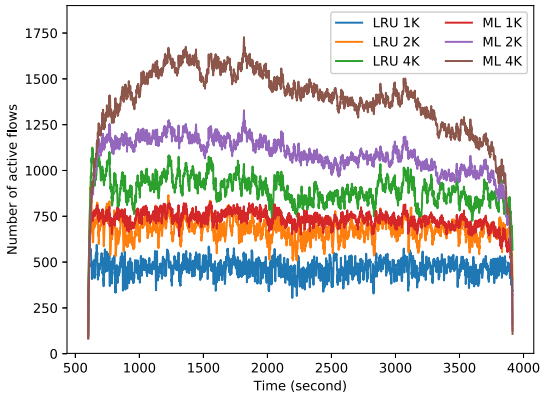

Fig. 10. Number of active flow entries in the flow table for the UNIV1 packet trace.

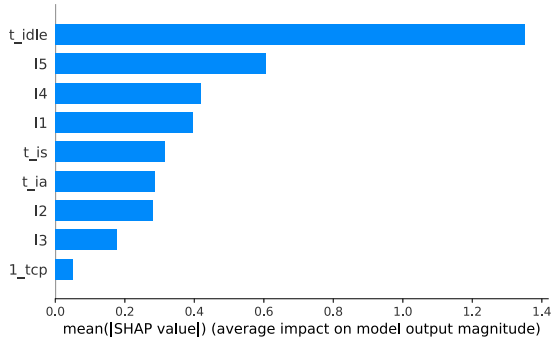

Fig. 11. Importance of GBT model features (`n_estimators` = 30; `subsample` = 0.8; `learning_rate` = 0.1; `max_depth` = 10) for UNIV1 trace with 1K flow table.

`learning_rate`:0.1; `max_depth`:10) with $N_{pkt} = 5$ and using 1 byte to quantize the time and packet length features. Although we cannot fully understand STEREOS' performance through this interpretation, it does provide some interesting insights. Our interpretations are based on the SHAP framework [31], which assigns each feature an importance (i.e., SHAP value) for a particular prediction such that the generated explanation model follows the definition of additive feature attribution methods and is subject to the properties of local accuracy, missingness, and consistency. The larger the SHAP value of a feature, the larger is the magnitude change in model output that results from the feature.

We first check the global mean of the absolute value of SHAP values for each feature, which are shown in Fig. 11.

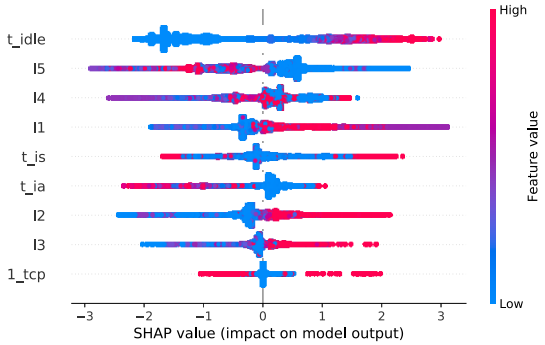

Fig. 12. SHAP values of GBT model features (`n_estimators` = 30; `subsample` = 0.8; `learning_rate` = 0.1; `max_depth` = 10) for every data sample (represented by one dot on each row), for UNIV1 trace with 1K flow table.

We can see that t_{idle} is the most important (1.35), which is the only feature used by the LRU policy. However, the following features (l_5, l_4, l_1, t_{is}) are also important – together their SHAP values are more than 1.74. To understand the importance of these additional features, we plot the SHAP values of every feature for each sample and the results are shown in Fig. 12. As can be seen, a small l_5 (the length of the last referring packet) indicates that the flow is more likely to be inactive while a large value indicates it is likely to be active. This is reasonable because the last packet of a flow is usually a signaling packet, which is short. For l_1 (the length of the first recorded packet), we see quite the opposite. Small l_1 is a good sign that the flow entry is active. In addition, we can see large t_{is} (standard deviation of the inter-arrival time of the last 5 recorded packets) indicates inactive flows. In summary, STEREOS learns other important features besides t_{idle}, which is the single feature used by LRU, and this allows STEREOS to more accurately classify active and inactive flows.

V. Conclusion

In this paper, we proposed machine learning techniques to optimize flow entry eviction in OpenFlow switches. We discussed implementation issues, including model selection, model size, overhead, and feature quantization. Case studies based on real network packet traces showed that the proposed techniques can achieve far fewer capacity misses and much higher flow table usage, compared with the widely-used LRU policy. Furthermore, network-level simulations demonstrate

that our techniques can greatly reduce control overhead, increase network throughput, and reduce packet loss rate.

REFERENCES

[1] A. Singh *et al.*, "Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 183–197, Aug. 2015.

[2] A. Greenberg, "SDN for the cloud," in *Proc. Keynote ACM Conf. Special Interest Group Data Commun.*, 2015.

[3] *OpenFlow Switch Specification (Version 1.5.1)*, Open Netw. Found. Std., Menlo Park, CA, USA, Mar. 2015.

[4] G. Lu *et al.*, "Serverswitch: A programmable and high performance platform for data center networks," in *Proc. Usenix NSDI*, 2011, p. 2.

[5] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp. 267–280.

[6] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski, "A knowledge plane for the Internet," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2003, pp. 3–10.

[7] C. Yu, J. Lan, Z. Guo, and Y. Hu, "DROM: Optimizing the routing in software-defined networks with deep reinforcement learning," *IEEE Access*, vol. 6, pp. 64533–64539, 2018.

[8] S. T. V. Pasca, S. S. P. Kodali, and K. Kataoka, "AMPS: Application aware multipath flow routing using machine learning in SDN," in *Proc. Nat. Conf. Commun.*, 2017, pp. 1–6.

[9] Y. He, F. R. Yu, N. Zhao, H. Yin, and A. Boukerche, "Deep reinforcement learning (DRL)-based resource management in software-defined and virtualized vehicular ad hoc networks," in *Proc. ACM Symp. Develop. Anal. Intell. Veh. Netw. Appl.*, 2017, pp. 47–54.

[10] C. Sieber, A. Obermair, and W. Kellerer, "Online learning and adaptation of network hypervisor performance models," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manage. (IM)*, May 2017, pp. 1204–1212.

[11] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho, "Deep learning approach for network intrusion detection in software defined networking," in *Proc. Int. Conf. Wireless Netw. Mobile Commun.*, 2016, pp. 258–263.

[12] Q. Niyaz, W. Sun, and A. Y. Javaid, "A deep learning based DDoS detection system in software-defined networking (SDN)," 2016, *arXiv:1611.07400*. [Online]. Available: https://arxiv.org/abs/1611.07400

[13] Q. Li, N. Huang, D. Wang, X. Li, Y. Jiang, and Z. Song, "HQTimer: A hybrid *Q*-learning based timeout mechanism in software-defined networks," *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 1, pp. 153–156, Mar. 2019.

[14] H. Yang and G. F. Riley, "Machine learning based proactive flow entry deletion for OpenFlow," in *Proc. IEEE Int. Conf. Commun.*, May 2018, pp. 1–6.

[15] K. Kannan and S. Banerjee, "Flowmaster: Early eviction of dead flow on SDN switches," in *Proc. Int. Conf. Distrib. Comput. Netw.*, 2014, pp. 484–498.

[16] H. Yang and G. F. Riley, "Machine learning based flow entry eviction for OpenFlow switches," in *Proc. Int. Conf. Comput. Commun. Netw. (ICCCN)*, 2018, pp. 1–8.

[17] R. Challa, Y. Lee, and H. Choo, "Intelligent eviction strategy for efficient flow table management in OpenFlow switches," in *Proc. IEEE NetSoft Conf. Workshops (NetSoft)*, Jun. 2016, pp. 312–318.

[18] T. Pan, X. Guo, C. Zhang, W. Meng, and B. Liu, "ALFE: A replacement policy to cache elephant flows in the presence of mice flooding," in *Proc. IEEE Int. Conf. Commun.*, Jun. 2012, pp. 2961–2965.

[19] B.-S. Lee, R. Kanagavelu, and K. M. M. Aung, "An efficient flow cache algorithm with improved fairness in software-defined data center networks," in *Proc. IEEE Int. Conf. Cloud Netw.*, Nov. 2013, pp. 18–24.

[20] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. Int. Conf. Passive Active Netw. Meas.*, 2015, pp. 347–359.

[21] Z. Guo *et al.*, "STAR: Preventing flow-table overflow in software-defined networks," *Comput. Netw.*, vol. 125, pp. 15–25, Oct. 2017.

[22] X. Lin, R. Blanton, and D. Thomas, "Random forest architectures on FPGA for multiple applications," in *Proc. Great Lakes Symp. VLSI*, 2017, pp. 415–418.

[23] T. Tanaka, R. Kasahara, and D. Kobayashi, "Efficient logic architecture in training gradient boosting decision tree for high-performance and edge computing," 2018, *arXiv:1812.08295*. [Online]. Available: https://arxiv.org/abs/1812.08295

[24] E. Alpaydin, *Introduction to Machine Learning*. Cambridge, MA, USA: MIT Press, 2014.

[25] C. Bergmeir and J. M. Benítez, "On the use of cross-validation for time series predictor evaluation," *Inf. Sci.*, vol. 191, pp. 192–213, May 2012.

[26] M. Dusi, F. Gringoli, and L. Salgarelli, "Quantifying the accuracy of the ground truth associated with Internet traffic traces," *Comput. Netw.*, vol. 55, no. 5, pp. 1158–1167, 2011.

[27] A. Vishnoi, R. Poddar, V. Mann, and S. Bhattacharya, "Effective switch memory management in OpenFlow networks," in *Proc. ACM Int. Conf. Distrib. Event-Based Syst.*, 2014, pp. 177–188.

[28] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.

[29] H. Yang, "Building scalable software defined OpenFlow networks," Ph.D. dissertation, Georgia Inst. Technol., Atlanta, GA, USA, 2019.

[30] M. Du, N. Liu, and X. Hu, "Techniques for interpretable machine learning," 2018, *arXiv:1808.00033*. [Online]. Available: https://arxiv.org/abs/1808.00033

[31] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 4765–4774.

Hemin Yang received the B.S. and M.S. degrees in computer and information science from Peking University, China, in 2012 and 2015, respectively, and the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology in 2019. He is currently a Software Engineer with Microsoft, Redmond, WA, USA. His research interests are in the area of software defined networking and machine learning applications.

George F. Riley received the Ph.D. degree in computer science from the Georgia Institute of Technology in 2001. He was a Professor of electrical and computer engineering with the Georgia Institute of Technology. He also worked on internet measurement methods, internet routing protocols, and software-defined networks. His research focused on creating more efficient methods and tools for simulation of both wired and wireless computer networks.

Douglas M. Blough received the B.S. degree in electrical engineering, and the M.S. and Ph.D. degrees in computer science from Johns Hopkins University, in 1984, 1986, and 1988, respectively. He is currently a Professor of electrical and computer engineering with the Georgia Institute of Technology. His research considers a variety of problems in computer networks with a primary interest in advanced wireless networking techniques.