2021 International Conference on Computer Communications and Networks (ICCCN) | 978-1-6654-1278-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCCN52240.2021.9522272

Machine Learning based Flow Classification in DCNs using P4 Switches

Radhakrishna Kamath and Krishna M Sivalingam
Dept. of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India
Email: cs18s030@smail.iitm.ac.in, skrishnam@cse.iitm.ac.in, krishna.sivalingam@gmail.com

***Abstract*—This paper deals with classifying flows in data center networks, primarily based on the flows' volume of traffic and duration. Flows are typically classified as long-lived flow or short-lived flow. Long-lived flows throttle the short-lived flows and should be classified at the earliest to select a different path in the network for them. The objectives of the proposed classification scheme are: (i) to support more than two flow classes (three in this paper), (ii) to achieve early classification by observing the first few packets in the flow, (iii) to achieve classification using ML techniques implemented in a programmable data plane switch using the Programming Protocol-independent Packet Processors (P4) language.**

Our contribution includes an improved hash-and-store algorithm for flow classification. The ML technique considered is Decision Tree, since it can be efficiently implemented in a P4 environment. The techniques have been evaluated using simulation-generated data implemented in a *mininet* emulator environment and classification accuracy results obtained. Two existing schemes, HashPipe and IdeaFix have also been implemented for comparison. The results show that the proposed scheme can classify a flow within 3 MB of the flow size when we consider more than one feature to classify the flows. This outperforms the existing threshold-based schemes by classifying flows, 3 times faster.

***Index Terms*—Datacenter networks, Flow Classification, Programmable data plane, Machine Learning, P4 language**

I. Introduction

Data centers have become an integral and important part of the computing infrastructure today. The efficient design of data center networks has gained much significance. One major consideration that helps drive network efficiency is the classification of traffic flows.

Traffic flows in the data center are mostly long-lived or short-lived. Long-lived flows are called *heavy-hitter* (or elephant) flows, and short-lived flows are called *small* (or mice) flows [1]–[4]. Studies have shown that the number of elephant and mice flows constitute about 10-20% and 80-90% of the overall flows in a data center [4], [5]. Elephant flows account for 80-90% of the traffic volume.

In data centers, Equal Cost Multipath (ECMP) routing is typically used [6]. ECMP only uses IP source address, IP destination address, transport source port, transport destination port, and transport protocol (five-tuple) to hash and decide a route without checking the link's load. This leads to the same link carrying multiple mice flows and elephant flows. Thus, elephant flows tend to dominate the mice flows and throttle them, by consuming a disproportional amount of buffer and link capacity. It is imperative to avoid such throttling in order to ensure that mice flows are not dropped or delayed inordinately. Thus, classifying (i.e. predicting) flows and handling them differently has been an active research problem for the past few years [7], [8]. Detecting the elephant flows can help to balance the load, increase the overall link utilization, identifying congestion and security attacks.

Early research on flow classification used techniques such as flow sampling and sketching (translating the stream characteristics in two-dimensional space) [1], [9], [10]. In recent years, with the advent of Software Defined Networking (SDN) [11], machine learning (ML) techniques were introduced in the SDN controller. Here, the switches sample flows and send these to the controller, which executes an ML algorithm for classifying the flows. Some of the common ML algorithms include Decision Tree, Support Vector Machine, Naive Bayes, and Deep Learning algorithms [12].

In all the above approaches, the goal of early detection was not achieved. Sampling needs to send around a large amount of data to classify, and even with ML in the controller, the controller will store the flow characteristics and then classify it. In this paper, one of the main objectives is to identify the flow type at the earliest, in the switch itself, using ML algorithms. The switch will have the ML inference engine, which will be trained offline. The switch will classify the flows and report the classification result to the controller. The controller will store the classification results.

Another objective of this work to explore programmable data plane (PDP) switches to implement the ML algorithms. Using the Programming Protocol-independent Packet Processors (P4) [13], [14], a data plane programming language, the data plane in switches can process a packet and take necessary action instead of acting as mere forwarding entities. The work presented in [15], [16] show that flow detection can be done in the data plane without involving the controller. Sketching algorithms that were implemented on a commodity switch were implemented in P4 for detecting the elephant flows at line rate. However, these do not guarantee early detection; for instance, the HashPipe scheme [15] reports the elephant flows after all the flows have passed through a switch.

Current-generation P4 switches do not support complex

This work was supported by a DST-FIST grant (SR/FST/ETI-423/2016) from Government of India and by a Mid-Career Institute Research and Development Award (IRDA-2017) from IIT Madras.

operations or floating-point computations since this will increase latency and complexity. Due to this limitation, ML algorithms were mapped to the match-action pipeline as shown in [17]. Here, these algorithms were trained offline and the inference engine was placed in the P4 switch. This leads to a tradeoff between accuracy and latency; that is, high-latency computations are avoided on the P4 switch which can result in lower classification accuracy.

In the proposed system, called SMArt P4 based SwitcH (SMASH), the ML algorithms are trained offline based on training data. The learned inference engine is implemented using P4 in the PDP switch, which is part of a Software-Defined Networking (SDN) environment. To classify the flows, the state of the flow, size of the flow, count of packets in the flow, inter-arrival time, and duration need to be stored. These values are stored in the switch using a sketch-based algorithm. Using the *P4Runtime* control plane API [14], the switches send the classification results to the controller, which records the same.

The novel features of the proposed system are: (i) Use of ML inference based switches to classify flows; and (ii) Design for early detection, that is, classify the flow with as few packets as possible in the beginning stages of a flow.

The proposed system has been implemented in P4 and Python. The PDP switch code is implemented in P4. The controller is implemented in Python and studied for 2-tier leaf-spine data center topology with network sizes up to four servers. Wisconsin data suite [4] has been used for analysis. The performance of the proposed system has been compared with HashPipe [15] and IdeaFix [18] algorithms, modified as needed. The results show that the proposed scheme can classify a flow within 3 MB of the flow size when we consider more than 1 feature to classify the flows. This outperforms the threshold-based schemes by classifying them 3 times faster.

II. Background and Related Work

This section presents the relevant background material and prior research. The proposed switch framework can be used in different network types: enterprise, data center, and core networks. In this paper, we are primarily considering data center networks (DCN) as an illustrative use case.

A. Data Center Networks

The data center is a centralized location that hosts computing infrastructure for storing, processing, and retrieval of large amounts of data. It consists of several hundred to thousands of computer servers where various applications are run. The servers are interconnected using a Data Center Network (DCN). The DCN architecture plays an important role in efficiently handling the traffic flows between the servers. The DCN is often hierarchical with multiple layers of switches, commonly referred to as access, aggregate, and core layers [19], [20].

Traffic flows in a DCN are characterized by many parameters such as the total volume of data generated, the total duration, the inter-arrival time between packets, and the average packet size. In order to achieve fairness among different flow types and to achieve high network utilization, different flows have to be treated differently. Typically, two types of flows are considered: heavy-hitter (elephant) and small (mice), as explained earlier. Studied in [4] show that around 80-90 percent of the flows are small in nature. However, the remaining ten-twenty percent of the flows generate most of the network traffic. ECMP routing, which is often used, routes all the flows without considering the flow characteristics. Thus, ECMP can route a heavy flow on the same path where one/many mice flows are traversing or where a heavy flow is traversing. Hence, detecting the heavy flows in order to handle them more effectively without affecting mice flows, is an important and active practical problem.

B. Flow Classification Techniques

One of the commonly used flow classification techniques is flow sampling, where the switches sample the flows and send this information to a central controller [1], [21], [22]. However, this requires a huge amount of data to be sent to the controller for statistical analysis. With the adoption of Software-Defined Networking (SDN) [11], the switches send the sampled flow information to the SDN controller, where detection and classification are done. There are different sampling rates like in 1 per 100, 1 per 1000, or 1 per 10000 packets. If sampling rate is low (e.g. 1 in 10,000), then the amount of flow already passed is huge and classification will be in error. If the sampling rate is high, then it will overwhelm the controller.

In [23]–[25], traditional machine learning (ML) and deep learning (DL) algorithms were used to detect the elephant flows. The controller stored the sample information until a threshold was reached, after which classification was done. This approach does not accommodate the changing nature of the network traffic since it is trained only once based on the past data and is not updated. In this scheme, early detection is not possible, since sampling is used. Another technique is called Sketching [9], [10], [26], which hashes and counts packets in switch hardware, for storing the flow characteristics in the switch. However, this incurs high memory overhead.

C. P4 switch based classification

The recent years have been significant development in P4, a programmable data plane language (PDP) [13] that can be used to realize flexible and programmable switches. Sketching techniquhas been combined effectively with P4 to implement classification algorithms in the switches. Hashpipe [15] is one such approach that does elephant flow detection in the switch. It keeps track of the k-heaviest flows in the switch by modifying an existing space-saving algorithm [27] into a d-stage sketching algorithm that keeps track of the packets of each flow. The algorithm uses d tables where the count of packets is stored. When a packet arrives, the five-tuple is extracted and looked up in the first table, and if present, the count is updated. Otherwise, the flow with the lowest count of packets is replaced and this lowest count flow will be sent to the next table. When a packet arrives, the five-tuple is extracted and looked up in the first table, and if present, the count is updated. Otherwise, the flow with the lowest count of packets is replaced and this lowest count flow will be sent to the next table. This will be done for all the flows passing through the switch. Once all the flows are passed through the switch, a memory mapping program will read the switch memory and get all the elephant flows stored by the switch. This algorithm's main limitation is that it never informs the controller about the flows, and all the elephant flows are detected after the flows are completed.

In IdeaFix [18], incoming flows are hashed into a set of registers, where the hash is computed based on the TCP/IP 5-tuple (Source Port and Address, Destination Port and Address, and the Protocol field). It uses a sketching technique to store the size, timestamp of the first packet, and the latest timestamp of the packet of all flows in the registers. In addition, a Bloom filter was used to determine whether a given flow has already been classified or not. It classifies a flow as an elephant type (or otherwise) using thresholds for cumulative byte count and active time of the flow and then informs the controller accordingly to perform suitable routing action. In the *mininet*-based emulator [28] experiments of an Internet Exchange Point (IXP) topology, the size threshold was set to 10 MB and the time threshold was 10 seconds. The average reaction time was shown to be significantly lower than that of the sFlow sampling scheme [21]. Since the scheme has to wait for 10 MB of data to pass through the switch, early detection was not possible.

The mechanism presented in [16] also stores the flow characteristics in the switch and periodically sends it to the controller. The controller then uses this information to classify flows and obtain a network-wide view instead of a single switch view. However, in our paper, we are not considering network-wide detection of elephant flows.

Some of the recent works deal with implementing the learned inference model in the P4 switch [17], [29], [30]. While trying to map the learned inference with the P4 match-action framework, there are several limitations due to P4's language capabilities. Here, match-action refers to a system where a rule tables contains two major entries: a set of conditions involving specific header fields and corresponding values of a network packet, and an action to be taken if an incoming packet matches the conditions present in a rule.

The work presented in [17] maps different ML techniques such as Decision Tree, Naive Bayes, Support Vector Machine, and K-Means Clustering to the match-action pipeline. It trains these algorithms offline using toolkits such as *scikit-learn* [31] and maps the learned inference to the match-action pipeline. There are different ways to map the learned inference to the match-action pipeline. One way is to have tables for every class with all the features as the table columns, another way is to have one table per feature. The decision tree is mapped to the match-action pipeline using tables for each feature. However, it does not provide the classification accuracy of the systems. It was implemented in the IoT security use case.

TABLE I
COMPARISON OF TECHNIQUES FOR CLASSIFYING FLOWS

Technique	**Early detection**	**ML-based**	**Detection in switch**
HashPipe [15]	N	N	Y
IdeaFix [18]	Y	N	Y
Harrison et al. [16]	Y	N	N
Viljoen et al. [25]	Y	Y	N
Xiao et al. [24]	Y	Y	N
Tong et al. [10]	Y	N	Y
SMASH (Proposed)	**Y**	**Y**	**Y**

Our proposed system combines several of these earlier ideas. First, machine learning techniques are implemented in the switch using P4, with training being done offline in the controller. Second, it uses the sketch-based approach to store the total size of the flow, count of packets of the flow, inter-arrival time between the packets of the flow, and the duration of the flow. Table I presents a comparison the existing schemes with the proposed scheme. As seen, the proposed system is the only one to consider early detection in the switch, using machine learning techniques.

D. ML Techniques in P4

There are several ML techniques that could be considered for implementation using the match-action pipeline of typical PDP switches and P4.

The Support Vector Machine (SVM) uses a hyperplane to separate the classes. Given k classes, there will be m tables, one per hyperplane, where $m = {}^{k}C_2$. If there is a match in a table, then a vote will be given to that class; the class with the highest votes will be reported.

The Naive Bayes scheme, where we have n probabilities and k classes, can be mapped to the match-action pipeline, using $k(n+1)$ tables. This is also not possible in P4, since it does not support floating-point values.

For K-means clustering, one way to map it to the match-action pipeline is to have a table for every cluster, with the key as the combination of all the features. The action for a match will be to find the distance from the core points of the cluster.

The *Decision Tree* approach [12], [32] is most suitable for implementation in the constrained P4 language. The Decision Tree (DT) has a tree-like structure that determines the class of the data point. It is mapped to the match-action pipeline using tables for each feature and depending on the feature matched in the table, the class will be determined using that label. It can be mapped to a match-action pipeline with higher accuracy since it mainly compares threshold values and does not perform any complex operation, as compared to other algorithms. Hence, we have considered this algorithm for implementation in the P4 switch.

III. Proposed System Design

This section presents the details of the proposed P4-switch based implementation of flow classification.

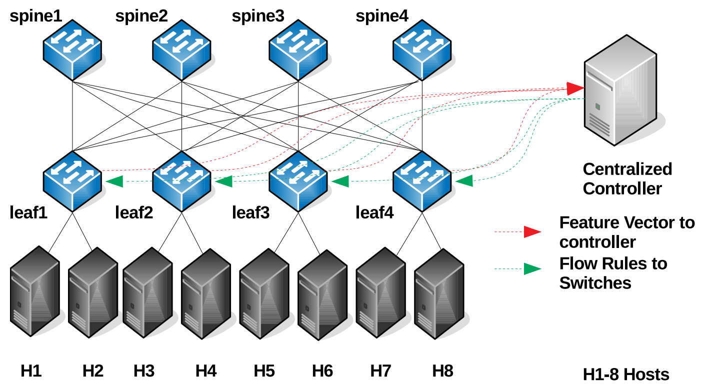


Fig. 1. System Architecture, consisting of P4-enabled smart switches in the leaf and spine levels, and an SDN controller.

TABLE II
TRAFFIC FLOW CLASSES

Traffic Class	Avg. Flow Size	Avg. Count	Avg. IAT (secs.)	Avg. Duration (secs.)	Application
0	30 KB	37	0.17	5.16	Transactions
1	650 KB	1666	0.90	425.11	Data/music file or interactive video, 10 minute video
2	3 MB	4291	1.73	414.07	Larger video/backup files, VM/Container migration

A. System Overview

For accurately classifying flows at the earliest possible time, classification should be done in the data plane and not in the control plane [15]. This is to avoid the latency due to sending flow information to the controller and to avoid sending a large number of packet information to the controller. To achieve this goal, we present the proposed SMASH (SMArt P4 based SwitcH) architecture, where the switches are data plane programmable using the P4 (or possibly other similar) language [13].

The proposed switch framework can be used in different network types: enterprise, data center, and core networks. In this paper, we are primarily considering data center networks (DCN) as an illustrative use case. Fig. 1 shows a two-tier DCN architecture, with servers (leaf nodes) interconnected using the proposed P4-enabled SMASH switches. A Software Defined Network (SDN) is considered, where the SDN controller interacts with the individual switches using a suitable South-Bound interface such as OpenFlow [33].

In this architecture, traditional compute servers run the necessary machine learning-based classification algorithms. The algorithms are trained using available network traffic data, that can be collected via network traces or using a simulated/emulated network. The inference models obtained from training are implemented in P4 to perform the flow classification in the SMASH switch. As described in Section II, the *Decision Tree* classification technique [12], [32] has been implemented in this work; other techniques such as Random Forest and Naive Bayes are also possible and will be considered in future work. When a switch classifies a flow as belonging to a specific class, it sends this information to the central SDN controller that generates the corresponding flow-handling rules and sends these to the switches. The learning threshold will be adjusted such that the stored flow information will be sent to the ML inference classifier at the earliest.

Typically, DCN flows are classified into an elephant (heavy-hitter) and mice (small) flows, with each flow type being routed using an appropriate mechanism. In this work, we have extended the granularity to three classes, for obtaining improved flow handling. Generally, size-based labeling is done. But this adds bias towards that feature (i.e. size) and the decision tree will use that feature only for making decisions. So to remove this bias, we used k-means, a clustering algorithm, to group the flows and then labeled each group as a class (three clusters). Four dimensions are used for clustering the data namely, the flow size, the number of packets in the flow, the inter-arrival time between packets in the flow, and the duration of the flow (4 dimensions). The average flow size, the average number of packets in the flow, the average inter-arrival time (IAT) between packets in the flow, and the average duration of the flow in each class are shown in Table II. It is also possible that there are other application classes that can fit these types. The rationale is that the finer granularity of classification can be used for enhanced buffer management and traffic scheduling. However, this is not considered in this paper and is left for future work to show the effectiveness of these multiple classes.

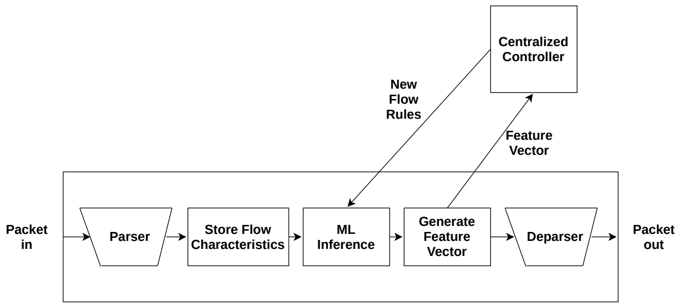


Fig. 2. SMASH switch components and the packet processing pipeline.

B. SMASH Switch Architecture

Fig. 2 presents the key components of the SMASH switch architecture. SMASH has a parser that parses the incoming packets, extracts the packet features from the header fields, and stores these. The features include the source and destination IP addresses, transport-layer port numbers, transport protocol, size of the packet, and packet arrival time.

Once a packet enters the switch, the following steps are executed:

a. Hashing and storing: The hash algorithm inputs are the different header fields (IP.src, IP.dst, Port.src, Port.dst, IP.protocol). The packet count, size, inter-arrival-time, and last-arrival time of a given flow are retrieved if it is already stored in the switch. These different header fields are then compared to the stored values of the flow. If there is a hash collision, then the stored fields like (IP.src, IP.dst, Port.src, Port.dst, IP.protocol, size of the flow, count of the packets, inter-arrival time, and duration of the flow) are sent for classification and new incoming header fields are updated in the registers. If there is no collision, and the stored size of the flow exceeds a certain threshold, then the values mentioned above are sent for classification. The threshold is a pre-decided value for filtering the very small mice flows from going into the classification module.

b. Classification: Classification of the flows is done using a machine learning algorithm, such as the Decision Tree technique used in this paper. Flow characteristics including the total byte count (size), packet count, packet inter-arrival time, and duration of the flow are compared with the values present in the switch table. The corresponding class will be assigned to the flow and sent to create a feature vector.

c. Feature vector creation and storage: After getting the values (IP.src, IP.dst, Port.src, Port.dst, IP.protocol, size of the flow, count of the packet, inter-arrival time, duration of the flow, and the class), a feature vector is generated and is sent to the controller. The controller will store the value and send these values to the traffic engineering module (not addressed in this paper). The traffic engineering module will create new forwarding rules and send to all the switches.

d. Updating the flow table: After receiving the forwarding rules, the switch will update the forwarding table.

The next section presents the details of the hash-and-store mechanism.

C. Hash-and-Store algorithm

The hash-and-store algorithm utilizes P4's features for hashing the values and the switch registers to store the flow characteristics. Since the number of hash registers in the switch are limited, an algorithm that uses the registers efficiently is needed. Three different cases are possible, as described below:

Case 1: As shown in Fig. 3, when a packet arrives, the IP five-tuple mentioned earlier, the size of the packet, and the ingress time_stamp (t) are obtained. The key (K) is calculated by hashing the five-tuple. This key K is matched with the key in the hash register. If the keys match, then the packet belongs to an existing flow.

The value of Last_time (Lt) is retrieved from the hash register at this key K and compared with the time_stamp retrieved. If this packet is within the time threshold, then packet count is incremented by one, and data size is incremented by the size of this packet. Start_time is not updated. Inter-arrival time is the time difference Lt and t. Last_time is replaced with t.

If the updated size exceeds the size threshold, all the values (start_time, last_time, inter-arrival time, size, and count) are sent for classification, and the packet is sent on the default route. After classification, the details will be sent to the controller and TE module. The controller will store the class label of that flow. The updated routing information obtained from the controller will be used for forwarding the flow.

Case 2: Consider the case where the hashed Key K does not match the key present in the hash register, indicating a hash collision. The processing is shown in Fig. 4. In this case, Last_time (Lt) and Start_time (St) are retrieved and replaced by t; Inter-arrival time by 0. Size by this packet's size; and count by 1. The remaining steps are as for the previous case, where the retrieved values are sent to the classifier and controller, and the updated forwarding rules for obtained from the TE via the controller.

Case 3: Consider an existing but inactive flow on which packets have not arrived for a long time, as specified by the time threshold. In this case, the Last_time (Lt) value is retrieved from the hash register at this key K. If (t-Lt) is above the time-threshold, then the old values are discarded and replaced by the new values. The packet is sent on the normal route. When enough packets are received for this flow, then the steps indicated in Case 1 will be followed.

D. ML Inference in Match-Action Pipeline

As explained earlier, when a flow's size threshold is crossed, its values are sent to the SDN controller for classification. In this work, we have mapped the *Decision Tree* algorithm [12], [32] to the match-action pipeline available in the P4 Switch.

The Decision Tree algorithm is run in the controller on the training data. Once the decision tree is trained, the tree is parsed and the conditions corresponding to the class label are retrieved.

There are different ways to map decision tree to the match-action pipeline. One way is to hard code the trained decision tree. But that will not have flexibility at all to retrain the model without recompiling the P4 code.

Two ways have been designed in this paper, to map the decision tree to the switch's match-action pipeline. Fig. 5 presents a possible decision tree for an example four-class system. Here, the internal nodes (represented by the conditions) will be used to map to the match-action pipeline. This method is developed to choose the conditions at a specific tree level with respect to the leaf nodes. Parsing and storing the entire tree will be complex and expensive, since the depth and the width of the tree can be very large. This can lead to a large number of entries pushed to the switch, and has to be avoided due to the space constraints for storing match conditions on the switch. To avoid these issues, we have developed two approaches:

SMASH-D1: This approach takes the conditions (internal nodes) that are one level above the labels (leaf nodes) and expands the values to ranges between the threshold and the average value for that feature in that condition. For example, Class 0, the condition is (size $< 14,000$) bytes; this will be expanded to a range of values between the average size of the

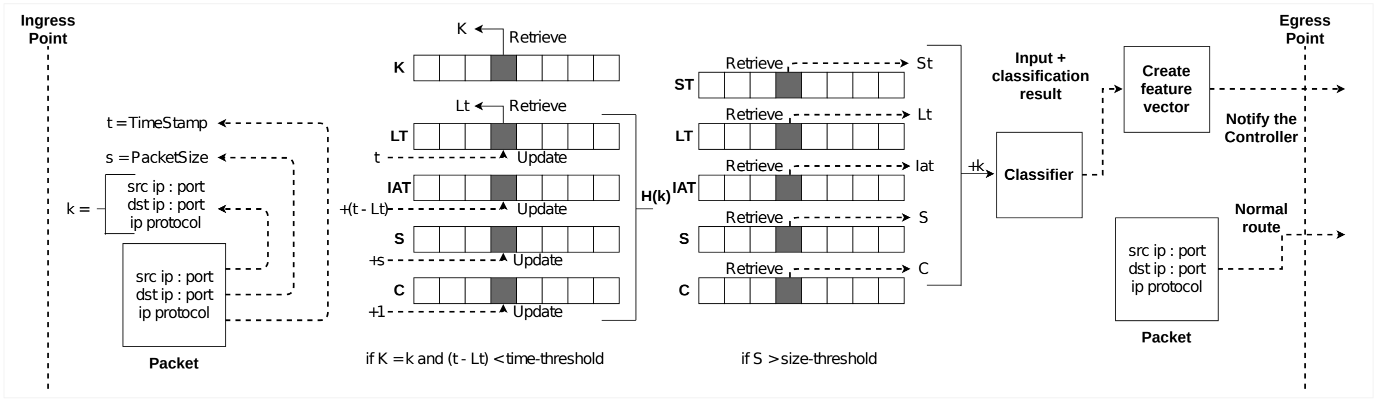


Fig. 3. Hash-and-store for a packet of an existing flow.

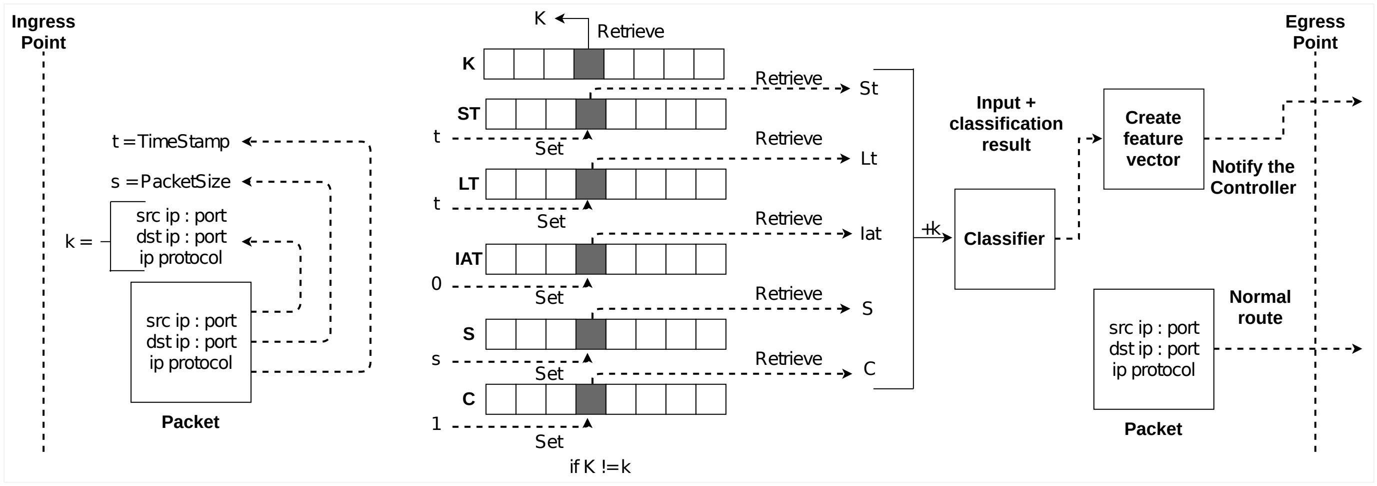


Fig. 4. Hash and store for packet of a new flow causing collision.

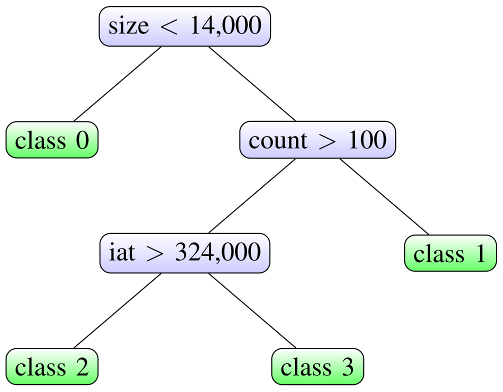


Fig. 5. Example of Decision Tree.

flows which belong to class 0 in the training data and 14,000, such that these values are in an arithmetic progression with a common difference. The common difference depends on the difference between the condition value and the average size of the flow. For instance, if 7,000 is the average flow size of class 0 flows, the range between 7,000 and 14,000 will be [7,000, 10,000, 13,000] with a common difference of 3,000. Each of these values will be expanded into a range of 1,500 bytes (MTU of Ethernet), since the flow sizes can be different and may not match a single value. For example, 7,000 will be expanded into a range of 1,500 values between 7,000 and 8,500. These 1,500 values will be pushed to the match table of the P4 switch. This way, a flow size between 7,000 and 8,500 is classified as class 0. The reason of taking few values is that the difference between condition value and the average size of the flow can be huge and it will overwhelm the switch with so many entries.

For class 1, the condition ($\text{count} > 100$) will be expanded into a range of values between the average number of packets of the flows which belong to class 1 and 100, such that the values are in an arithmetic progression with a common difference. In our study, we have used 60 as the average number of packets in class 1 flows. For example, the values of [60,75,90] are pushed to the match table of the P4 switch and used for matching packet count values.

For class 2 and class 3 condition related to packet inter-arrival time, ($\text{IAT} > 324,000$) microseconds, IAT is expanded into a range of values between the average IAT value of the class 2 flows and the average IAT of the class 3 flows. For instance, let 289,000 be the average IAT for the class 3 flows and 345,000 as of the average IAT for class 2 flows. The range will be from 289,000 to 345,000, since the inter-arrival time between packets can be different and may not match a single value. Thus, a value of 289,000 to 324,000 will label the flow as class 3 and a value of 324,000 to 345,000 will label flows to class 2. These values are in microseconds range and hence can require a large number of match entries in the table. To reduce this requirement, each value in the range is divided by 1,024 and in the switch, it is equivalently right-shifted by 10 bits. The final list which would be inserted in the table would look like [282,316] for class 3 and [317,336]] for class 2. Similarly, the flow duration value is divided by 65,536 and right-shifted by 16 bits in the switch.

SMASH-D2: This approach takes the conditions (internal nodes) that are two levels above the labels (leaf nodes) and expands the top condition into ranges between the threshold and the average value for that feature in that condition. For class 0, there is only one condition $\text{size} < 14,232$, which is expanded into a range of values as explained above.

For class 1, there are two conditions $(\text{size} > 14,000)$ and $(\text{count} < 100)$. Since the condition on size is above the other condition in the tree, it is expanded into a range of non-consecutive values between the average flow size of the flows belonging to class 1 and 14,000, and as described above. The second condition will be parsed in the action part. When there is a match in the size table, the count of the packets will be checked. If it is less than 100 packets, then the flow is classified as a class 1 flow.

For class 2 and class 3, the two conditions at the bottom of the tree will be considered, namely $(\text{count} > 100)$ and $(\text{IAT} > 324,000)$. Here, the first condition will be expanded to a range of non-consecutive values between the average number of packets in class 3 flows and 100. Thus, for any count value matching the table entry, the IAT will also be checked. If it is greater than 324,000, then the flow is classified as a class 2 flow, and class 3 otherwise.

This section presented the details of the proposed SMASH architecture for P4-based switches; the performance evaluation results are presented in the next section.

IV. Performance Evaluation

This section presents the performance study based on the implementation of the proposed SMASH switch architecture.

A. Implementation Details

Dataset: The Wisconsin Data Suite [4] was used and the flows labeled using the K-means clustering algorithm, as described in Section III. Once labeled, the data set had around 326,484 flows with 32,649 (10%) flows as *outliers* (302,996 flows belong to class 0, 22,340 flows belong to class 1, and 1,148 flows belong to class 2). Of this, 80% of this flow data set was used for training and 20% for testing. The training was done for the decision tree algorithm using the *scikit-learn* package [31]. The features used to train were: size_of_the_flow, count_of_packets, inter-arrival_time, and duration_of_the_flow. The testing data is divided into 50 batches, and for each batch, we calculated the metrics and calculated the mean and standard deviation for calculating 95% confidence interval.

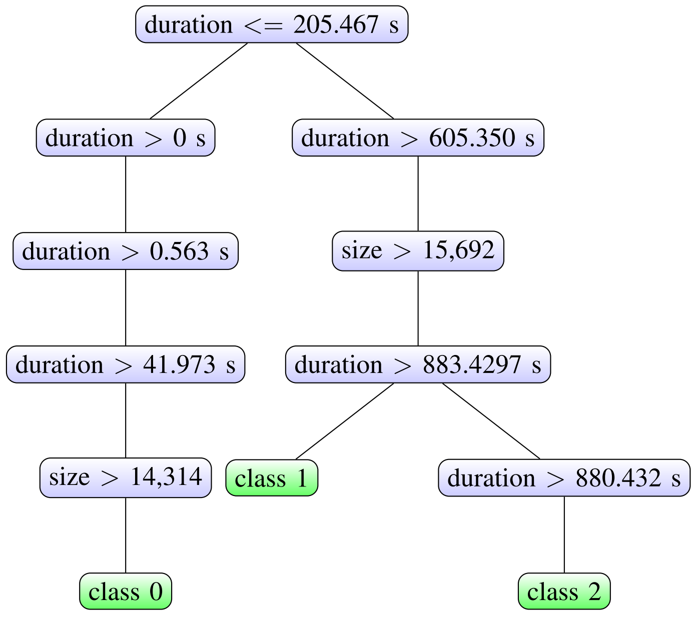


Fig. 6. Decision tree generated from the training data.

Fig. 6 shows the decision tree trained from the training data. This tree is parsed and table entries are created. For SMASH-D1, internal nodes such as: (i) size greater than 14,314 bytes, (ii) duration greater than 880.432 s, and duration greater than 883 s are used as explained in Section III-D. For SMASH-D2, internal nodes such as: (i) duration $>$ 41.973 s, size greater than 14,314 bytes; (ii) duration less than or equal to 883.429 s and duration greater than 880.432 s, (iv) size greater than 15,692 bytes and duration greater than 883.429 s are used. The first condition is expanded for the table entries and the second condition is sent to the action block.

Mininet emulator: The SMASH switch architecture has been implemented in *mininet*, a network emulator [28]. The P4 programs are compiled with a target of BMV2 (behavioral model version 2), an open-source software switch developed by the P4 team [34]. Mininet was run on a Dell server that had 32 cores and 2 threads per core, with a clock speed of 1202 MHz and 128 GB RAM. The studied data center network (DCN) topology has multiple leaf switches and multiple spine switches connected in a mesh network. In Fig. 1, a system with four hosts, four leaf switches, and four spine switches is shown. These *mininet* hosts generate and exchange packets, based on the studied data set (explained later). A standalone Python program is used to realize the SDN controller; it updates the flow entries using Mininet's *P4Runtime* mechanism to all the switches. The entire implementation is available online [35]; it can be decrypted with MZEEbUBayJpypR7YSC5h as the password: *unzip -P* *password Smash-project.zip -d Smash-project/*

The test set from the Wisconsin data set was divided into fifty batches, as mentioned above. Once the decision tree is trained, it is parsed and the appropriate table entries are created and pushed to the P4 switch tables in *mininet* using P4Runtime commands, by the SDN controller. Each *mininet* host generates packets using *scapy* package [36], using these flow characteristics.

B. System Parameters and Metrics

The *DecisionTreeClassifier* implementation of the *scikit-learn* package provides some tunable hyper-parameters. These include: (i) the criterion used to measure the quality of a split (*Gini* and *entropy*); (ii) the splitter strategy used to choose the split at every node (options are *best* or *random*), and (iii) the max_depth of the tree (values of 5, 10, 15). In our experiments, off-line 5-fold cross-validation was done to select the set of hyper-parameters that provided the highest mean accuracy: the *Gini* criterion with *best* splitter strategy and 10 as the max_depth.

The number of hosts was varied as 8 and 12 hosts, with 4 leaf switches and 4 spine switches. The number of flows generated using mininet was also varied. We have studied the performance for 200 flows (which take around 3 hours of emulation time to complete), and 500 flows (takes around 8 hours). The experiments are run with 40% flows of class 0, 30% flows of class 1, and 30% flows of class 2. A total of eight scenarios were considered by varying number of hosts (8 and 12), number of flows (200 and 500), and decision-tree mapping variants (SMASH-D1 and SMASH-D2).

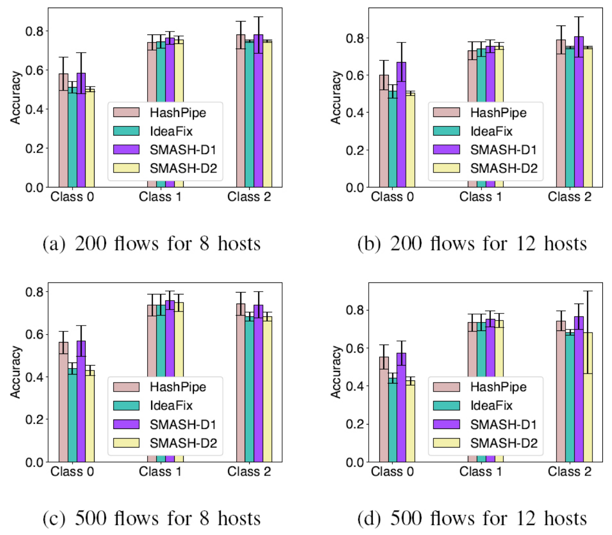
earliest classification

The main objective of the proposed system is to classify the flow accurately and at the earliest. One metric is the total byte-count of a given flow that has passed through the switch before classification is done. This value should be small and at the same time, the flow classification has to be correct, as measured using the standard *accuracy* (ratio of sum of true positives and true negatives to the entire data set size), *precision* (ratio of true positives to the sum of true positives and true negatives), and *recall* (ratio of true positives to the sum of true positives and false negatives) metrics [12].

C. Results and Analysis

The SMASH system is compared to two existing schemes, HashPipe [15] and IdeaFix [18]. Since the implementation of the schemes was not provided and these only differentiate 2 classes of flows, we have implemented them with slight modifications. HashPipe is modified to send the flows after 1 MB (class 1) and 10 MB (class 2) of data is passed through the switch, since the original algorithm does not send the flow classification information to the controller. IdeaFix uses a Bloom filter to maintain the list of currently classified flows. If all the bits in the Bloom filter are set, then the unclassified flow will also be marked as classified and not informed to the controller. To avoid this, we have modified the algorithm and added a periodic reset of the Bloom filter. Further, IdeaFix will also send the flow information to the controller at 1 MB for class 1 flows and at 10 MB for class 2 flows.

The test data used for each of the experiments is a different subset of the same test data. HashPipe, IdeaFix, and SMASH-D1, and SMASH-D2 ran on the same test data for a particular scenario.



(a) 200 flows for 8 hosts (b) 200 flows for 12 hosts

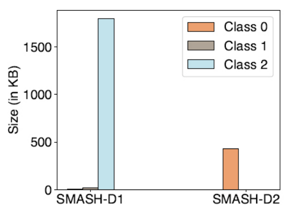
(c) 500 flows for 8 hosts (d) 500 flows for 12 hosts

Fig. 7. Comparison of accuracy achieved using HashPipe, IdeaFix with SMASH-D1 and SMASH-D2 schemes.

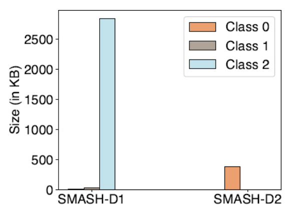
Fig. 7(a) shows the classification accuracy results for SMASH-D1, SMASH-D2, HashPipe, and IdeaFix for 200 flows with 8 hosts. HashPipe has an accuracy rate of 58% for class 0 flows, 74% for class 1 and 78% for class 2 flows. IdeaFix has an accuracy rate of 51.3% for class 0 flows, 74.5% for class 1 and 74.8% for class 2 flows. IdeaFix fails in classifying the class 1 flows and this accuracy is because of misclassification of class 1 flows as class 0 flows. SMASH-D1 has an accuracy rate of 58.5% for class 0 flows, 76.5% for class 1 and 78% for class 2 flows. SMASH-D2 has an accuracy rate of 50.3% for class 0 flows, 75.5% for class 1 and 74.8% for class 2 flows. Here, SMASH-D1 has nearly same accuracy as HashPipe and it performs better than IdeaFix. SMASH-D2 does not perform well and performs similar to IdeaFix.

Fig. 7(b) shows the classification accuracy results for SMASH-D1, SMASH-D2, HashPipe, and IdeaFix for 200 flows with 12 hosts. Here, SMASH-D1 has performed slightly better than HashPipe and much better than IdeaFix. Fig. 7(c) and Fig. 7(d) show a similar trend for 500 flows with 8 and 12 hosts respectively.

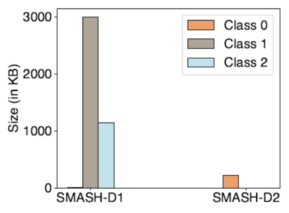
Fig. 8 presents the flow size (in KB) required by SMASH-D1 and SMASH-D2 before classifying the flows. The figure shows that SMASH-D1 can classify class 0 flows within 4 KB, in all the scenarios. It identifies class 1 flows within 20 KB for 200 flows but requires 3 MB of data, for the case of 500 flows. This happens because the flows need to have a flow duration above 883 secs for getting classified as class 1. If the average number of flows has a lower inter-arrival of time, then a higher amount of traffic passes in that duration. If the inter-



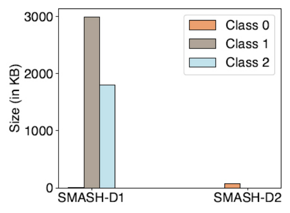
(a) 200 flows for 8 hosts



(b) 200 flows for 12 hosts

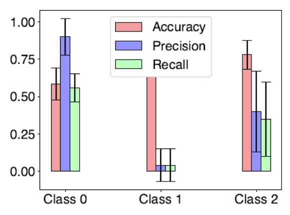


(c) 500 flows for 8 hosts

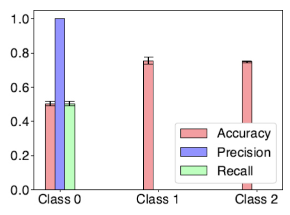


(d) 500 flows for 12 hosts

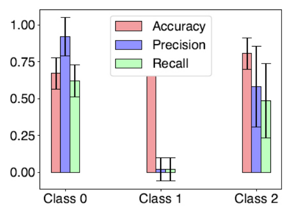
Fig. 8. Flow sizes required by SMASH-D1 and SMASH-D2 for classifying the flows.



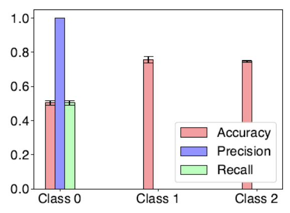
(a) SMASH-D1: 8 hosts



(b) SMASH-D2: 8 hosts



(c) SMASH-D1: 12 hosts



(d) SMASH-D2: 12 hosts

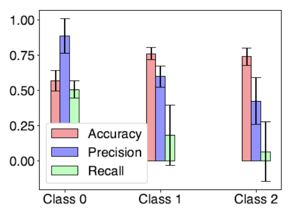
Fig. 9. Classification results comparison of SMASH-D1 and SMASH-D2 for 200 flows and varying the number of hosts.

arrival time is large, then only some amount of traffic flows during this duration. It identifies class 2 flows within 1 MB in scenarios like 200 flows with 12 hosts and 500 flows with 8 hosts. It needs 1.8 MB for the remaining class 2 scenarios.

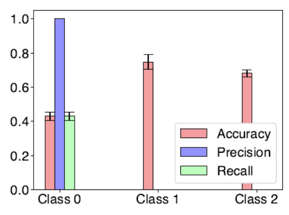
SMASH-D2 identifies class 0 flow within 429 KB in all the scenarios. However, it could not identify class 1 and 2 flows. This is due to the fact that there are two conditions to check unlike SMASH-D1, with one condition in the match table and another in the action block. Only if both are satisfied is the class label assigned. Thus, it is seen that the accuracy of HashPipe, IdeaFix, SMASH-D1, and SMASH-D2 are nearly the same; however, flows can be classified within 3MB by SMASH-D1 thus detecting the flows 3 times early (since HashPipe and Ideafix required 1 MB and 10 MB of data to classify class 1 and class 2 flows).

Fig. 9(a) presents accuracy, precision and recall results of SMASH-D1 and SMASH-D2 for 200 flows and 8 hosts. SMASH-D1 classifies class 0 with 58.5% accuracy, class 1 with 76.5% accuracy, and class 2 with 78% accuracy. The precision of SMASH-D1 for class 0 is 90%, which means out of the total flows classified as class 0, 90% of them are correct. The recall value of SMASH-D1 for class 0 is 56%, which implies that only 56% of the class 0 flows were classified. For class 1 flows, SMASH-D1 has a precision of 4% and recall of 4%. For class 2, SMASH-D1 has precision of 40% and recall of 34.7%.

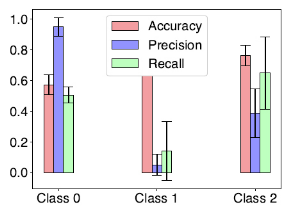
Fig. 9(b) shows that SMASH-D2 achieves accuracy of 50.3% for class 0 flows, 75.5% accuracy for class 1 flows and 74.8% accuracy for class 2 flows. Precision and recall for class 1 and class 2 flows is 0%, which means SMASH-D2 is not at all classifying the class 1 and class 2 flows. It is misclassifying all the flows as class 0 flows. For class 0 flows, the precision is 100% but only 50.3% of the class 0 flows are getting classified. Similar trends are seen in Fig. 9(c) and Fig. 9(d).



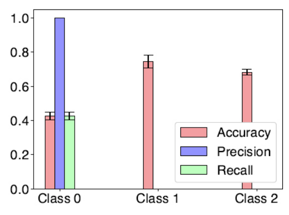
(a) SMASH-D1: 8 hosts



(b) SMASH-D2: 8 hosts



(c) SMASH-D1: 12 hosts



(d) SMASH-D2: 12 hosts

Fig. 10. Comparison of SMASH-D1 and SMASH-D2 for class 0, 1, and 2 for 500 flows and varying the number of hosts.

Fig. 10(a) presents accuracy, precision and recall results for SMASH-D1 and SMASH-D2 for class 0, 1, and 2 for 200 flows for 8 hosts. SMASH-D1 can classify class 0 with 56.8% accuracy, class 1 with 76% accuracy, and class 2 with 73.8% accuracy. The precision of SMASH-D1 for class 0 is 88.5%, that is, 88.5% of class 0 flows are correctly classified. Recall of SMASH-D1 for class 0 is 50.5%, which means only 50.5% of the class 0 flows were classified. For class 1 flows, SMASH-D1 has a precision of 6.3% and recall of 18%. For class 2, SMASH-D1 has precision of 42.3% and recall of 59.7%.

Fig. 10(b) shows that SMASH-D2 achieves accuracy of 43% for class 0 flows, 74.8% accuracy for class 1 flows and 68.2% accuracy for class 2 flows. Precision and recall for class 1 and class 2 flows is 0%, which means SMASH-D2 is not at all classifying the class 1 and class 2 flows. It is misclassifying all the flows as class 0 flows. For class 0 flows, the precision is 100% but only 43% of the class 0 flows are getting classified. Similar trends are seen in Fig. 10(c) and Fig. 10(d).

In summary, the experiments have showed that the proposed SMASH-D1 performs better than SMASH-D2 in terms of classification metrics and in terms of lower flow-size requirements. Also, SMASH-D1 is better than the existing HashPipe and IdeaFix schemes in terms of lower flow-size requirements, while achieving similar or better classification performance.

V. Conclusions

This paper has presented an approach called SMASH to classify the DCN flows in the switch using the ML inference at an early stage. It has also shown that SMASH can classify three types of flows and with nearly the same accuracy as the threshold-based schemes but classifying them as early as 3 MB of the flow size. The results show that adding the ML inference in the switch enhances the switch thus enabling it to classify the flows at the earliest.

This work can be further enhanced in several ways. For instance, feedback about mis-classified flows can be collected by the switch after sufficient amount of data has been processed; this feedback can be provided to the controller to update the classification model. Also, a TE module that computes suitable routes for the different traffic types can be developed. Finally, the architecture can be realized using a hardware platform such as the Xilinx NetFPGA SUME board. Also, different hyperparameters for training the Decision Tree can be tried to obtain a better model with higher accuracy. Other ML algorithms, such as Random Forest and Naive Bayes can also be implemented in P4 switches for detailed comparisons.

Acknowledgments The authors gratefully acknowledge the comments and suggestions received from Dr. Vyas Sekar of CMU, and Dr. Arkadeep Sen and Mr. Harsh Gondaliya, formerly of DAWNLab, CSE Dept., IIT Madras.

References

[1] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, "Identifying elephant flows through periodically sampled packets," in *Proc. of ACM SIGCOMM Internet Measurement Conference*, 2004, p. 115–120.

[2] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements and analysis," in *Proc. of ACM Internet Measurement Conference (IMC)*, 2009, p. 202–208.

[3] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. of ACM SIGCOMM*, 2015, pp. 123–137.

[4] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. of ACM Internet Measurement Conference (IMC)*, 2010, p. 267–280.

[5] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," *Proc. of ACM SIGCOMM Internet Measurement Conference (IMC)*, vol. Part F1319, pp. 78–85, 2017.

[6] C. Hopps, "RFC 2992: Analysis of an Equal-Cost Multi-Path Algorithm," Nov. 2000. [Online]. Available: https://tools.ietf.org/html/rfc2992

[7] R. Trestian, G. Muntean, and K. Katrinis, "Micetrap: Scalable traffic engineering of datacenter mice flows using openflow," in *Proc. IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2013, pp. 904–907.

[8] F. Carpio, A. Engelmann, and A. Jukan, "DiffFlow: Differentiating Short and Long Flows for Load Balancing in Data Center Networks," in *Proc. of IEEE GLOBECOM*, 2016, pp. 1–6.

[9] C. H. Liu, A. Kind, and A. V. Vasilakos, "Sketching the data center network traffic," *IEEE Network*, vol. 27, no. 4, pp. 33–39, 2013.

[10] D. Tong and V. Prasanna, "High throughput sketch based online heavy hitter detection on FPGA," *SIGARCH Comput. Archit. News*, vol. 43, no. 4, pp. 70–75, 2016.

[11] "Software-Defined Networking (SDN) Definition," 2021. [Online]. Available: https://opennetworking.org/sdn-definition/

[12] C. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[13] The P4 Language Consortium, "P4$_{16}$ Language Specification," 2020 Jun. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.1.pdf

[14] "P4runtime," 2020. [Online]. Available: https://p4.org/p4-runtime

[15] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. of ACM Symposium on SDN Research*, 2017, p. 164–176.

[16] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches," in *Proc. of ACM Symposium on SDN Research (SOSR)*, 2018, pp. 1–7.

[17] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *Proc. of ACM HotNets*, Jan. 2019, pp. 25–33.

[18] M. V. B. da Silva, A. S. Jacobs, R. J. Pfitscher, and L. Z. Granville, "IDEAFIX: Identifying Elephant Flows in P4-Based IXP Networks," in *Proc. of IEEE GLOBECOM*, 2018, pp. 1–6.

[19] Cisco, "Cisco data center infrastructure 2.5 design guide," Dec. 2013.

[20] B. Wang, Z. Qi, R. Ma, H. Guan, and A. V. Vasilakos, "A survey on data center networking for cloud computing," *Elsevier Computer Networks*, vol. 91, no. C, pp. 528–547, Nov. 2015.

[21] "sFlow: Making the network visible," Nov. 2020. [Online]. Available: https://sflow.org/

[22] Cisco Systems, "Cisco IOS Flexible NetFlow," 2007. [Online]. Available: https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html

[23] Y. Afek, A. Bremler-Barr, S. L. Feibish, and L. Schiff, "Sampling and Large Flow Detection in SDN," in *Proc. of ACM SIGCOMM*, 2015, pp. 345–346.

[24] P. Xiao, W. Qu, H. Qi, Y. Xu, and Z. Li, "An efficient elephant flow detection with cost-sensitive in SDN," in *Proc. of International Conference on Industrial Networks and Intelligent Systems (INISCom)*, 2015, pp. 24–28.

[25] N. Viljoen, H. Rastegarfar, M. Yang, J. Wissinger, and M. Glick, "Machine learning based adaptive flow classification for optically interconnected data centers," in *Proc. of International Conference on Transparent Optical Networks (ICTON)*, 2016, pp. 1–4.

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

[27] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. of Intl. Conf. on Database Theory (ICDT)*, 2004, pp. 398–412.

[28] "Mininet," Nov. 2020. [Online]. Available: https://mininet.org

[29] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the AI accelerator?" in *Proc. of Workshop on In-Network Computing (NetCompute)*, 2018, pp. 20–25.

[30] G. Siracusano and R. Bifulco, "In-network neural networks," *CoRR*, vol. abs/1801.05731, 2018.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[32] P.-N. Tan, M. Steinbach, A. Karpatne, and V. Kumar, *Introduction to Data Mining*. Pearson, 2019.

[33] Open Networking Foundation, "OpenFlow Switch Specification Version 1.5.1," Mar. 2015. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[34] "Working with P4 in Mininet on BMV2," Nov. 2020. [Online]. Available: https://build-a-router-instructors.github.io/deliverables/p4-mininet/

[35] R. Kamath, "SMASH P4 Code," Apr. 2021. [Online]. Available: https://www.cse.iitm.ac.in/~skrishnam/Smash-project.zip

[36] "Scapy: Packet crafting for Python2 and Python3," Apr. 2021. [Online]. Available: https://scapy.net/