pHeavy: Predicting Heavy Flows in the Programmable Data Plane

Xiaoquan Zhang, Lin Cui, *Member, IEEE,* Fung Po Tso, *Senior Member, IEEE* and Weijia Jia, *Fellow, IEEE*

***Abstract*—Since heavy flows account for a significant fraction of network traffic, being able to predict heavy flows has benefited many network management applications for mitigating link congestion, scheduling of network capacity, exposing network attacks and so on. Existing machine learning based predictors are largely implemented on the control plane of Software Defined Networking (SDN) paradigm. As a result, frequent communication between the control and data planes can cause unnecessary overhead and additional delay in decision making.**

In this paper, we present *pHeavy*, a machine learning based scheme for predicting heavy flows directly on the programmable data plane, thus eliminating network overhead and latency to SDN controller. Considering the scarce memory and limited computation capability in the programmable data plane, *pHeavy* includes a packet processing pipeline which deploys pre-trained decision tree models for in-network prediction. We have implemented *pHeavy* in both bmv2 software switch and P4 hardware switch (i.e., Barefoot Tofino).Evaluation results demonstrate that *pHeavy* has achieved 85% and 98% accuracy after receiving the first 5 and 20 packets of a flow respectively, while being able to reduce the size of decision tree by 5.4x on average. More importantly, *pHeavy* can predict heavy flows at line rate on the P4 hardware switch.

***Index Terms*—Heavy flow, Decision tree, Programmable data plane, P4**

I. Introduction

There is always a need for network operators to analyze network traffic for improving network reliability, performance, configuration and security management. Heavy flows, which account for a significant fraction of network traffic [36], have profound impact on networks as they either can cause network congestion or indicate an ongoing DoS attack. Therefore, identifying heavy flows correctly and promptly is of great importance to many applications such as mitigating link congestion [14], scheduling of network bandwidth [22] and exposing network attacks [30].

Existing solutions for identifying heavy flows can be broadly categorized as detection or prediction based. Good heavy flow detectors/predictors need to consider tradeoff among three key metrics: *accuracy*, *timeliness* and *network overhead* [37]. This means that the accuracy needs to be high enough for the predictors to be meaningful. The prediction needs to take place as early as possible to give network administrators sufficient early response time. For example, FlowSeer [16] proved that prediction in early response time can effectively enhance the performance of load balancers. Network overhead also should be minimized to avoid impact on normal network traffic.

Statistics-based detection schemes (e.g., Hashpipe [36], PRECISION [12]) collect statistics for monitored flows. However, having a fixed threshold means they are not flexible enough to cope with the dynamism of network traffic. A long detection time is usually required before counters exceeding the fixed threshold. Another way for detection is to sample only a fraction of packets of target flows. However, since monitoring overhead is inevitable in statistics collection, tradeoff between sample frequency (which affecting *accuracy*) and communication *overhead* is still needed.

In comparison to detection, machine learning can predict heavy flows early with high accuracy using only the first few packets of a flow. This gives extra headroom for network operators to take actions on the heavy flows. Most existing machine learning methods need to leverage the SDN architecture [16] [26]. The data plane collects traffic features and sends them to the controller, which runs machine learning algorithms to predict heavy flows. However, such approaches introduce unnecessary or even heavy communication overhead and network delays due to the additional communication between the data plane and controller, violating *timeliness* and *network overhead* attributes above.

Clearly, achieving all three aforementioned metrics simultaneously is challenging. To overcome this challenge, in this paper, we exploit the data plane programmability and propose *pHeavy*, a machine learning approach for predicting heavy flows in the programmable data plane. Our results show that *pHeavy* can predict heavy flows with 85% accuracy upon receiving only the first 5 packets of a flow. Moreover, by predicting entirely in the data plane, *pHeavy* eliminates unnecessary communication overhead and network delays between data plane and controller.

It is challenging to train accurate machine learning models that are simple enough to be implemented in the programmable data plane. First, the number of heavy flows is usually a very small fraction of total flows albeit being accountable for most network packets transferred [26]. Such imbalanced distribution can mislead the construction of machine learning models, producing inaccurate and large-sized models consuming more scarce memory of switches. Second, programmable data planes usually have limited memory and computation power, which limits statistics collection and the deployment of

Xiaoquan Zhang and Lin Cui are with the Department of Computer Science, Jinan University, Guangzhou, China. Email: zhangxiaoquan547@gmail.com, tcuilin@jnu.edu.cn.

Fung Po Tso is with the Department of Computer Science, Loughborough University, UK. Email: p.tso@lboro.ac.uk.

Weijia Jia with BNU-UIC Institute of Artificial Intelligence and Future Networks, Beijing Normal University (BNU Zhuhai) and BNU-HKBU United International College, Zhuhai, China. Email: jiawj@uic.edu.cn.

Corresponding author: Lin Cui

prediction algorithms in the programmable data plane[1] [39].

In short, this paper has made the following contributions:

1) A training algorithm is proposed to eliminate effects of imbalanced distribution to obtain accurate models and reduce the size of decision trees for data plane implementation.
2) Considering limited computing resources in programmable data plane, a packet processing pipeline is designed to track each flow's features and predict heavy flows, and a flow management scheme is also enforced to optimize memory usage.
3) A prototype on P4 hardware switch (Barefoot Tofino) is implemented. Evaluation results show that *pHeavy* can predict heavy flows accurately at line rate.

The rest of paper is organized as follows. Section II gives an overview of related works and explains challenges for *pHeavy* design. Section III presents the system overview of *pHeavy*. Section IV and V describe the detailed *pHeavy* design, including network traffic data training and predicting entirely in the programmable data plane. Section VI discusses the implementation details of hardware and software switches, Section VII shows the evaluation results and Section VIII concludes the paper.

II. Related works & Challenges

A. Related works

Recent research works on identifying heavy flow can be divided into two categories (see Table I): statistics-based detector and ML-based predictor.

Statistics-based detector: These schemes are based on flow statistics to identify heavy flows. Sampling is a commonly used method [20], which assumes that heavy flows are more likely to be tracked since they generate most of network traffic. Probability sampling [34] can be used to reduce the complexity of detection. However, in order to achieve high accuracy, the sampling process must collect as much flow information as possible. This could introduce high overhead between the data plane and controller. Another way of detecting heavy flow is to periodically collect information, e.g., Hedera [11] and Helios [21]. Similarly, such periodically pulling of statistics of flows may introduce significant network overhead (e.g., 1 to 20MB signaling overhead in data center network [16]) in order to provide precise accuracy. Other works, namely heavy hitter (e.g., Hashpipe [36], DevoFlow [17] and [13]), set up counters in the data plane to detect heavy flow. They count the number of packets transferred in each monitored flow. Although they are easy to implement in switches, they require a long detection time until the counter exceeds a pre-defined fixed threshold. Overall, methods based on statistics can not detect heavy flows in the early stage since they need to wait until the heavy flow bursts.

ML-based predictor: Some works adopt machine learning algorithms to predict heavy flow. They are usually implemented in the controller of SDN networks. Pouper et al. [33] discussed predicting heavy flow by three machine learning algorithm: neural networks, Gaussian process regression and online Bayesian Moment Matching. Xiao et al. [38] utilized a cost-sensitive model in decision trees to train models with high accuracy in heavy flow prediction. FlowSeer [16] designed a two-phase method. The first phase uses cost-sensitive decision trees and the second phase implements data mining with the Hoeffding tree in the controller. Huang et al. [26] proposed a heavy flow decision scheme that consists of two stages. The first stage uses $C4.5$ decision tree [35] and the second stage adopts a more accurate machine learning algorithm APPR [25]. However, by deploying ML-based predictors on the controller, they inevitably introduce signaling overhead and delay between the controller and data plane.

B. Challenges

Two important challenges must be considered in actual design and implementation of an machine learning based predictor for heavy flows. First, imbalanced distribution of heavy flows can lead to an inappropriate model during training. Second, programmable hardware data planes usually have limited memory and computation capability. This means any resulting models will need to be lightweight for them to be run on the hardware data plane.

1) Imbalanced data problem in heavy flow prediction: In practice, network traffic contains about 10% of heavy flows. Interestingly they carry more than 90% of total packets [37]. *Such extreme imbalanced distribution will mislead the classifier, causing unacceptable accuracy*. This is called imbalanced data problem in machine learning [24]. Since most algorithms assume balanced class distributions or equal misclassification costs, these algorithms will fail to represent the distributive characteristics of the data and result in low prediction accuracy across the classes of imbalanced data. For example, $C4.5$ decision tree uses IGR (Information Gain Ratio) to select decision features and determine branches, and features with high IGR can be used to discriminate different classes well. However, this mechanism does not consider the accuracy of classification of the minority class. Furthermore, these solutions may lead to large-sized models (Experiments in Section IV-C), which can not be implemented in the programmable data plane with limited memory (e.g., TCAM).

2) Offloading in the data plane: Programmable switches usually have limited memory and computation power due to the cost of high-speed hardware. This creates challenges for the implementation of *pHeavy*. Scarce memory (e.g., few tens of megabytes) in a switch will limit the size of machine learning models and the number of flows it can monitor. Some optimization techniques, such as removing inactive flows and their associated flow status from the flow tables, could be used to free up more memory, but programmable data planes currently lack dynamic memory management to support such optimization. Furthermore, most machine learning algorithms (e.g., SVM or neural network) cannot be implemented in the programmable data plane due to the requirement of floating operations. Moreover, some frequently-used statistics operations such as multiplication/division and average are also difficult to implement.

[1]For example, Barefoot Tofino [4] only supports 12 stages in the pipeline and dozens of megabytes of available memory, and does not support multiplication and division.

Table I
COMPARISON OF *pHeavy* WITH OTHER SOLUTIONS

Type	ML-based		Statistics-based	
Location	Data plane	Controller	Data plane	Controller
Method	Decision tree	Decision tree & Hoeffding tree	Heavy hitter	Simple & probability sampling
Examples	*pHeavy*	FlowSeer [16] Huang et al. [26] Xiao et al. [38]	Basat et al. [13] DevoFlow [17] Hashpipe [36]	Netflow [20] SIFT [34]
Detection/prediction delay	Low (e.g., 90% of flows within 1.2 seconds in UNI1 dataset)	Prediction & communication delay	High (e.g., 90.3% of flows within 2.0 seconds in UNI1 dataset)	High
Switch-controller communication overhead and delays	None	Moderate	None	Heavy
Accuracy	High (e.g., 20 packets with 97.6% TNR in UNI2 dataset)	High	Low (e.g., 20 packets with 79.5% TNR in UNI2 dataset)	Low

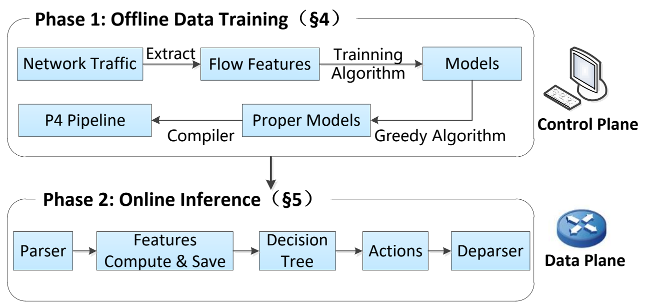


Figure 1. The overview of *pHeavy*

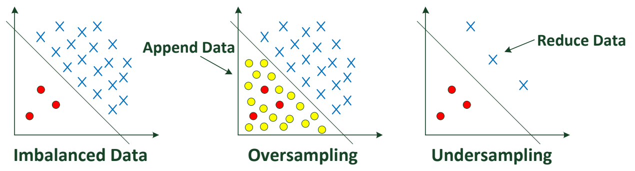


Figure 2. Two sampling methods. Oversampling appends data to be balanced dataset and undersampling reduces data to be balanced dataset.

III. SYSTEM OVERVIEW

This section provides an overview of *pHeavy* which consists of an offline model training and online inference as illustrated in Figure 1.

Offline Model Training: In this phase, the controller trains machine learning models and compiles them into components of target switches which enables prediction at runtime in the second phase. *pHeavy* uses real network datasets [1] containing network information in the transport and network layers (e.g., in PCAP format) as the input in the first phase, extracts features of each flow from the dataset and labels each flow in advance.

Afterwards, *pHeavy* utilizes its training algorithm to train models, in which the undersampling method randomly generates different training datasets that can train models with different performance. It provides more options to achieve a tradeoff between the size of models and accuracy. Since undersampling will produce multiple trained models with different performance, a greedy algorithm is also designed to determine appropriate models with better performance that can be deployed to the data plane. Lastly, the selected decision trees will be compiled for target switches.

Online inference: The second phase is predicting heavy flow in the programmable data plane. *pHeavy* uses decision trees because of their simplicity in components (e.g., conditional statements) and operations (e.g., does not need floating), which can be easily implemented in the programmable data plane. The *pHeavy* pipeline firstly extracts packet's header by the parser module for computing flow features, and then saves them in registers. When the number of received packets of a flow reaches a preset value, e.g., 5 packets, the flow will be verified by a pre-installed decision tree and tagged with a flag (e.g., heavy or non-heavy flow). After a flow is judged as a certain type, its subsequent packets will not modify its features and network operators can apply their desired actions based on the requirements of network applications.

IV. NETWORK TRAFFIC TRAINING

This section focuses on how to train machine learning models for *pHeavy*.

A. Heavy flow and metrics

Heavy flow: A flow f_i is a sequence of packets with the same 5-tuple (source IP, destination IP, source port, destination port, protocol). M_i is denoted as the total length of packets of f_i. A subflow is denoted as $f(i:j)$ and subsequently the first j packets of a flow are denoted by $f(1:j)$.

Definition IV.1. Given a data stream consisting of a set of flows F, a flow $f_i \in F$ is defined as a *heavy flow* with respect to a customized threshold ϕ:

$$M_i \geq \phi$$

And, the *occupation rate* of heavy flow in F is defined as:

$$\alpha = \frac{|\{f_i | M_i \geq \phi, f_i \in F\}|}{|F|} \quad (1)$$

Confusion matrix and metrics: Identifying heavy flows can be seen as a binary classification, where the 0 class represents non-heavy flow and 1 class is heavy flow. Given the two-class case, a confusion matrix is shown in Table II.

Table II
THE CONFUSION MATRIX OF BINARY CLASSIFICATION

	1	0
1	True Positive (TP)	False Negative (FN)
0	False Positive (FP)	True Negative (TN)

True Positive (TP) means the observation is positive, and the sample is predicted to be positive. False Negative (FN) is that the observation is positive, but the sample is predicted to be negative. True Negative (TN) describes that observation is negative, and the sample is predicted to be negative. And False Positive (FP) represents that observation is negative, but the sample is predicted to be positive.

Two metrics are used to evaluate the accuracy: true positive rate (TPR) and true negative rate (TNR) [33]. TPR is the ratio of the total number of correctly classified positive samples to the total number of positive samples, and TNR is the ratio of the total number of correctly classified negative samples to the total number of negative samples:

$$TPR = \frac{TP}{TP + FN}$$

$$TNR = \frac{TN}{TN + FP}$$

Thus, TPR and TNR represent the percentage of heavy flows that are correctly predicted and the percentage of non-heavy flows that are correctly predicted, respectively [33]. Therefore, *the objective of pHeavy is to achieve high TPR and TNR.*

B. Network traffic features

There are many useful network traffic features revealed from previous works [32]. Considering limitations of the programmable data plane, *pHeavy* only selects features that are feasible to be implemented in the programmable data plane. There are two types of features: stateless and stateful features, as shown in Table III. Stateless features refer to intrinsic characteristics of a flow (e.g., destination TCP/UDP port). And stateful features should be saved and computed in the programmable data plane (e.g., total length).

Table III
NETWORK TRAFFIC FEATURES

Name	**Type**	**Description**
IAT_min	Stateful	Minimum of packet inter-arrival time
IAT_max	Stateful	Maximum of packet inter-arrival time
IAT_avg	Stateful	Average of packet inter-arrival time
IAT_total	Stateful	Total of packet inter-arrival time
len_min	Stateful	Minimum of packet length
len_max	Stateful	Maximum of packet length
len_avg	Stateful	Average of packet length
len_total	Stateful	Total of packet length
SYN/ACK	Stateful	TCP SYN/ACK flag counter
PSH/ECE/RST	Stateful	TCP PSH/ECE/RST flag counter
sport/dport	Stateless	Source/Destination port

Table IV
PERFORMANCE ANALYSIS ON IMBALANCED DATA(METRICS ARE INTRODUCED IN SECTION IV-A AND VII-A)

10% of heavy flow in UNI2			
	Cost-sensitive	Undersampling	Oversampling
TPR	**0.834**	**0.854**	0.663
TNR	0.760	0.760	0.881
F-measure	0.537	0.546	0.567
Tree size	969	**273**	1083
2% of heavy flow in UNI2			
	Cost-sensitive	Undersampling	Oversampling
TPR	0.379	**0.838**	0.415
TNR	0.958	0.781	0.960
F-measure	0.287	0.180	0.295
Tree size	979	**81**	1011

C. Imbalanced data problem

Imbalanced data problem model: Assuming an imbalanced distribution consists of a minority class and a majority class. Considering a given dataset S, two subsets $S_{min} \subset S$ and $S_{maj} \subset S$ are defined. S_{min} is the set of minority samples in S and S_{maj} is the set of majority samples in S. And, $|S_{maj}| \gg |S_{min}|$ and $|S| = |S_{maj}| + |S_{min}|$.

Cost-effective learning and sampling: The basic idea of cost-sensitive learning is the concept of cost matrix. Cost matrix numerically represents the penalty of classifying samples from one class to another. Assuming a binary classification, the cost matrix can be defined as:

$$C(i, j) = \begin{pmatrix} 0 & c \\ d & 0 \end{pmatrix}$$

where d is denoted as the cost of misclassifying a majority class (in heavy flow problem, setting d to 1 represents no penalty for false negative), and c is denoted as the cost of misclassifying a minority class. The objective of cost-sensitive learning is to minimize overall cost on training data, and it changes the class i of each flow f to the class j such that $\sum_i P(i|\boldsymbol{S})C(i, j)$, where $P(i|\boldsymbol{S})$ represents the probability of each class i for a given training dataset $\boldsymbol{S}$. Subsequently, tuning a cost matrix value can obtain a favorable model for training data.

Sampling methods aim to balance the number of minority class and majority class. Undersampling removes samples of the majority class (e.g., random-based [3]). So,

$$|S_{new}| = |S_{min}| + |S_{maj}| - |S_{under}| \tag{2}$$

where S_{new} is defined as the new data set after sampling and $S_{under} \approx S_{maj} - S_{min}$ is the data set that is reduced from the majority data set. Oversampling appends samples to the minority class (e.g., clustering-based [24]). So,

$$|S_{new}| = |S_{min}| + |S_{maj}| + |S_{over}| \tag{3}$$

where $S_{over} \approx S_{maj} - S_{min}$ is the data set that is appended to minority data set. Figure 2 illustrates difference of the two solutions. The Equation (3) implies that the new constructed dataset will produce more complicated machine learning models, which is difficult to be executed in the data plane. In contrast, the reduced amount of the dataset in Equation (2) enables more simple machine learning models.

Data trace driven analysis: To investigate the performance of above methods, a trace driven analysis is conducted using a real network dataset (i.e., UNI2 [1]). Information of each flow in the dataset is analyzed and more than ten common network features are computed based on the first 20 packets of each flow. The cost-sensitive learning and two sampling methods are used to predict heavy flows under two different occupation rates, i.e., 2% and 10% [16].

Three common machine learning metrics and the size of decision trees are used to evaluate, as shown in Table IV. Results show that only undersampling method has the highest TPR with the smallest tree size. However, its tree size is still too large to be implemented in programmable data plane. Furthermore, when the occupation rate of heavy flows is reduced from 10% to 2%, both TPR and F-measure of all methods are decreased, which indicates that neither of these methods can independently handle imbalanced distribution of heavy flow effectively. Notice that the cost-sensitive method can obtain high TPR. Thus, *pHeavy* adopts a training scheme combining the undersampling and cost-sensitive (e.g., meta-cost [18]) methods as the fundamental algorithm to overcome the imbalanced data problem. Experiments show that, after going through several decision trees built by different undersampling training data, *pHeavy* can achieve the same accuracy as the method based on controller.

D. Decision tree training

The training algorithm of *pHeavy* is depicted in Figure 3. S_i is defined as dataset consisting of features of subflows ($f(1:i)$), and S_i^T and S_i^P are defined as original training dataset and verifying dataset respectively splitted from S_i. s_i^T is subset of S_i^T after sampling. T_i is defined as a decision tree that predicts heavy flows after receiving the i^{th} packet. Four datasets are defined to represent results of the decision tree T_i with tagging class 0 or class 1: $S_i^T{}_{(0)}$, $S_i^T{}_{(1)}$, $S_i^P{}_{(0)}$, and $S_i^P{}_{(1)}$, where $S_i^T{}_{(0)}$ and $S_i^T{}_{(1)}$ are prediction results of S_i^T, and $S_i^P{}_{(0)}$ and $S_i^P{}_{(1)}$ are prediction results of S_i^P.

The whole training process consists of multiple stages as shown in Figure 3. Each stage includes several steps. Initially, original dataset is divided into original training dataset and verifying dataset. The former would be reduced to the balanced training dataset (e.g., s_i^T) through the two undersampling methods, i.e., OSS (One-Sided Selection) [29] and Random [3]. The OSS selects a representative subset of the majority class to combine it with the minority class as a new training dataset, which removes samples that have similar characteristics with the sample of the majority class to compact the majority class. The Random undersampling equitably balances the amount of the majority class samples and the minority class samples. Each randomly selected sample can only represent parts of characteristics of the majority class, so that it generates simple models with low TNR. Leveraging the property of Random undersampling, *pHeavy* lets each flow go through one or several simple machine learning model(s) for high TPR and TNR. Then, the balanced training dataset is used to train a decision tree (e.g., T_i) with high TPR and low TNR by the cost-sensitive algorithm. In other words, the decision tree aims to filter non-heavy flows but maintain as many heavy flows as possible. Next, in each prediction step, the decision tree will give two predicted results: class 0 (non-heavy flow) or class 1 (heavy flow). The dataset with class 0 will be classified as non-heavy flow, and the dataset with class 1 will become a new prediction dataset (e.g., S_j^T) for the next stage. To increase accuracy of subsequent training models, *pHeavy* also predicts the training data (e.g., S_i^P) and utilizes the results as the new training dataset (e.g., S_j^P) in next stage. It will delete the flows whose characteristic has been integrated in the decision tree, so that the Random undersampling method would not select them in the next stage.

Algorithm 1 Greedy searching algorithm

```
Input: $Q$, the number of tree
       $thr$, threshold of tree score
       $S^T = \{S_i^T, S_j^T, ...\}$, training dataset
       $i, n$, define the starting and stopping search location
       $E$, the searching pace
Output: $L$, list of result consisting of decision trees
 1: $Starter \leftarrow i + 1$
 2: $m \leftarrow 0$                              //initialize number of tree
 3: $L \leftarrow \emptyset$                              // initialize list of result
 4: while $m < Q$ && $Starter < n$ do
 5:     $m \leftarrow m + 1$
 6:     $j \leftarrow Starter$                        // move $j$ to search
 7:     $T\_list \leftarrow \emptyset$                        // save training result
                                            // - - - search satisfied tree
 8:     while $score\_temp < thr$ do
 9:         $j \leftarrow j + 1$
10:         $T\_list \leftarrow$ Train($S_j^T$)                // training process
11:         $(T\_temp, score\_temp) \leftarrow$ Max($T\_list.score$)
12:     end while
                                            // - - - save satisfied tree
13:     $L$.insert(0,$T\_temp$)
14:     $Starter \leftarrow j$
                                            // - - - expand searching range
15:     $j_e \leftarrow E$
16:     while $j_e > 0$ do
17:         $j_e \leftarrow j_e - 1$
18:         $j \leftarrow j + 1$
19:         $T\_list \leftarrow$ Train($S_j^T$)
20:         $(T\_temp, score\_e) \leftarrow$ Max($T\_list.score$)
21:         if $score\_e > score\_temp$ then
22:             $L$.pop(0)
23:             $L$.insert(0,$T\_temp$)
24:             $Starter \leftarrow j$
25:         end if
26:     end while
27: end while
```

E. Selecting decision trees

The Random undersampling method will produce multiple datasets to train decision trees with different performance. To obtain better performance with limited resource consumption in the data plane, *pHeavy* adopts a greedy searching algorithm

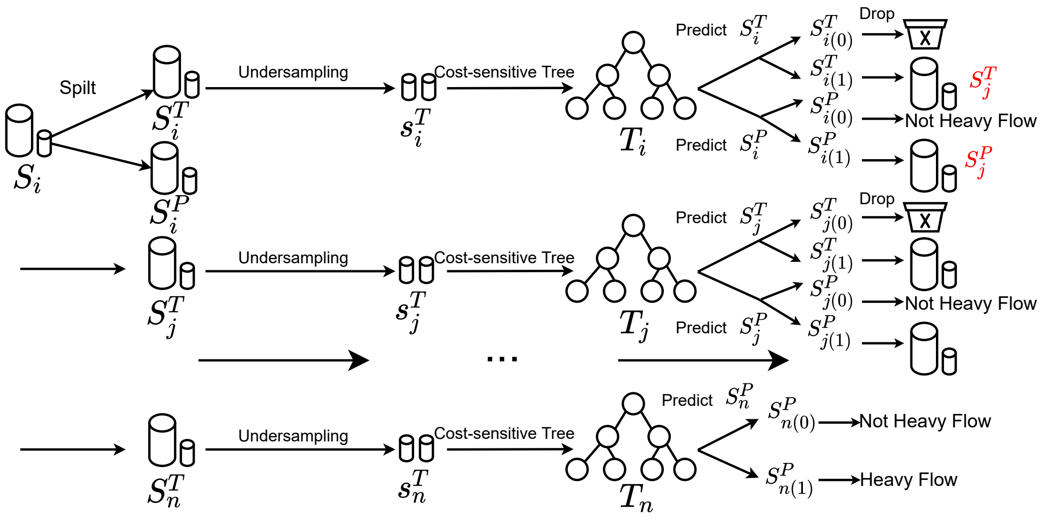


Figure 3. The training algorithm consists of several stages, each of which generates a decision tree, a training dataset and a predicted dataset for next stage (marked by red), until the final stage determines heavy flow

(see Algorithm 1) to select several optimized decision trees in certain locations (e.g., i^{th} packet).

Criterion for tree selection: *pHeavy* selects a decision tree based on three conditions: small size, high TPR and high TNR. A score is calculated based on these three conditions for all decision trees. Then, a tree will be selected if its score exceeds a predefined threshold.

Greedy searching algorithm: A greedy algorithm is designed to select specific number (e.g., Q) of decision trees satisfying the criterion (e.g., thr) in a searching range (e.g., i to n) as early as possible. Our primary experimental analysis has shown that small interval between predicting locations of two decision trees may decrease the performance of the later trees. To address the problem, a variable E is used as the pace to expand searching range to improve performance when a satisfied tree is found.

To show the effectiveness of the greedy algorithm, we also implement a random method for decision tree selection, which will randomly select four predicting locations and each location has a trained decision tree.

V. Prediction in data plane

This section explains how online inference works and addresses limitations in the data plane of P4 switch.

A. Online inference in the pipeline

The processing pipeline of online inference in the programmable data plane is illustrated in Figure 4.

Parser: The parser component parses packet headers which are used to extract and compute features, e.g., TCP flags. This information will traverse with the packet to following stages.

Ingress & Egress: A packet will be firstly hashed to a corresponding flow and then be processed differently: (i) if the packet is attached with termination TCP flags or it exceeds IAT limit, its corresponding flow's memory space will be initialised (e.g., set to 0); (ii) if the flow has been categorized as a heavy flow, predefined actions (e.g., driving flows to leisure links for load balancing [16]) will be applied; (iii) the update of corresponding flow's features and prediction by a decision tree (e.g., T_5) will be triggered when it satisfies predefined conditions (e.g., the arrival of the 5^{th} packet). In the end of the pipeline, all packets need to be deparsed for packet construction before their departure.

Computing components: *pHeavy* uses 5 tuples of a packet, i.e., {source IP address, destination IP address, source port, destination port, protocol}, to uniquely identify a flow by hash algorithms which P4 supports. In addition, P4 switch offers timestamps when the packet enters the ingress pipeline, which can be used to compute the time related variables (e.g., IAT).

All features are stored in registers and computed by basic operations supported by P4 (e.g., addition, subtraction, hash). For example, the value of ACK flag counter will be increased if the ACK flag of an incoming packet is set. Since P4 does not support division operation, *pHeavy* uses exponentially weighted moving average (EWMA) to implement average operation [15]. The EWMA can be represented as:

$$S_t = \begin{cases} y_1, & t = 1 \\ \alpha \cdot y_t + (1-\alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

Let $\alpha = 0.5$, which can be implemented through bit shift.

B. Compilation of decision tree in the P4 switch

P4 allows metadata traversing through different stages in the pipeline. Once prediction is triggered, *pHeavy* fetches the values of features from registers and saves them in metadata as inputs of the decision trees in following stages.

There are two ways to realize decision trees in the data plane: (i) Decision trees can be implemented in the control

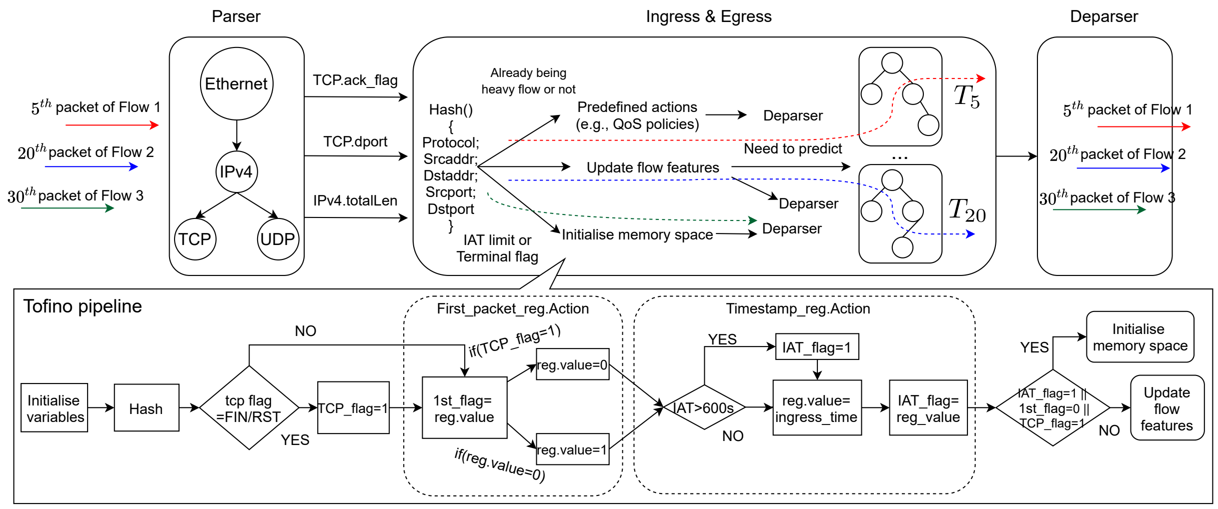


Figure 4. *pHeavy* 's pipeline and the process of flow management in the P4 hardware switch

flow block using variables and the conditional statement (i.e., *if* and *else*). Variables store the value of nodes of the decision tree, and the function of conditional statements resemble branches. For example, assuming a node *dport* > 1234, the variable saves the value 1234 and pHeavy uses *if* and *else* to control different executions. (ii) Each root node of a decision tree is a result of prediction, and it is destined by all features of the decision tree which have specific value ranges. For example, the prediction result of a node is a heavy flow, and the values of all features satisfying the result are $A \in [a_1, a_2]$ and $B \in [b_1, b_2]$, etc. This process is similar to the flow table in P4 switch using range matching. Each entry in the table has match fields and actions. The match fields can be represented as features' value range and the actions will set the prediction result of flows (e.g., flag). Thus, a decision tree can be transferred into a table with different entries in the pipeline. The larger the decision tree, the more table entries are required, thereby consuming more memory (e.g., TCAM).

A flow needs to go through one or several decision trees in order to reach an identification (e.g., $T_i, ..., T_n$). In other words, except for the final decision tree, all intermediate decision trees do not identify any heavy flows, but can make decisions on non-heavy flows. Hence, *pHeavy* uses three flags to tag it. Flag 0 means a flow is undetermined and needs continuous monitoring. Flag 1 and flag 2 represent non-heavy flow and heavy flow respectively. Figure 4 gives an example of three distinct flows that traverse the pipeline. For each flow, the verification of each decision tree happens only once when the switch receives the i^{th} packet of the flow. For example, the 5^{th} packet of flow 1 (red dashed line) and the 20^{th} packet of flow 2 (blue dashed line) are predicted by T_5 and T_{20} respectively.

C. Memory management

Since the number of registers is much smaller than the number of flows, *pHeavy* adopts a memory management strategy to allocate memory dynamically.

There are two rules indicating termination or expiration of a flow. Considering a TCP flow sending signal packets to inform connection termination (e.g., FIN flag and RST flag), *pHeavy* exploits these TCP flags as the sign of termination of a flow. Upon receiving a packet with such termination flags, a flag of its corresponding row space in the register of the flow will be set, indicating the register is released. On the other hand, real network trace [1] shows that some flows only transfer few packets without any termination TCP flags. Therefore, *pHeavy* uses interval arrival timeout as another signal of flow termination (i.e., *pHeavy* sets IAT>600 seconds). Due to the lack of packets with termination flags, an expiration flag can only be tagged by other flows which hash to the same slot. Once hash collision happens and the register space is full, a new flow can reuse the memory space of the original flow that has been expired or terminated.

VI. Implementation

We have implemented *pHeavy* on bmv2 [5] consisting of over 700 lines of $P4_{16}$ code (including several decision trees with maximum depth 10), and P4 hardware switch (Flnet S9180-32X) with a 3.2Tb/s Barefoot Tofino 32D ASIC [4].

A. Offline model training

UNIBS dataset: The UNIBS dataset [9] [19] [23] were collected on the edge router of the campus network of Brescia University during three consecutive working days. We use the day3 traffic trace which is mainly composed of TCP (99%) and UDP to evaluate *pHeavy*.

UNI dataset: The UNI dataset [1] has two packet traces from two university data centers, UNI1 and UNI2. The major traffic in UNI1 is TCP traffic. By contrast, most heavy flows are UDP traffic in UNI2. Although the number of TCP flags would not be useful in the UDP flow, experiments in UNI2 show that *pHeavy* also performs well in UDP flows.

Model training: The model training is completed in Weka 3 [10]. *pHeavy* uses dpkt [2] to extract features of network traffic and RandomUnderSampler [3] to mitigate imbalanced data problem. scikit-learn [7] is used to split dataset into training dataset (70%) and testing dataset (30%) for the method "holdouts" validation [27] [31].

Threshold: The value of a threshold will affect the number of flows that are recognized as heavy flows, and hence impacts the accuracy of prediction. However, there is no unanimous value for the threshold. For example, Helios [21] defined heavy flow with flow rate below 15Mbps, and Devoflow [17] defined three thresholds with different values (128KB, 1MB and 10MB). To satisfy the needs of different networking environments, *pHeavy* allows network operators setting the threshold, e.g., occupation rate α (Section IV-A), for heavy flow according to their requirements.

B. pHeavy in the P4 hardware switch

Following aspects are considered when implementing *pHeavy* on a Barefoot Tofino P4 hardware switch.

Register operation: In P4 hardware switches, all operations on the same $Register$ must be in the same stage. For example, in the update of a $Register$ used to save the feature value, it may require several operations: reading the value, calculation and writing a new value. *pHeavy* adopts the $RegisterAction$ block provided by $P4_{16(Tofino)}$ to aggregate all operations for a $Register$ in the same stage.

Features: Since Barefoot Tofino only offers limited operations, *pHeavy* only uses a small part of flow features, as shown in Table III. In the experiment, it was found that the prediction has close accuracy with prediction in the software switch and the size of decision trees is only marginally increased (e.g., 5% in UNIBS dataset). Table V lists the top three important features in each decision tree based on information gain ratio [35], showing which feature is more important in the prediction. In dataset UNI1, TCP flags are important features in software switches (bmv2) and P4 hardware switches (Tofino). In comparison, UNI2 prefers length and IAT features since the dataset mainly consists of UDP traffic.

Memory management in P4 hardware switch: There are two major challenges for implementing memory management in Barefoot Tofino: (i) Due to the limited number of stages (e.g., 12 stages) in the pipeline, *pHeavy* needs to use as few stages as possible to save space for other applications. (ii) Although $RegisterAction$ allows programmers to access $Register$ multiple times, the $RegisterAction$ block can not be split into sub-blocks for using at different stages.

To overcome the two challenges, *pHeavy* defines two $Registers$ and three flag variables. The two $Registers$, i.e., $First\_packet\_reg$ and $Timestamp\_reg$ are shown in the two dotted boxes in Figure 4. $First\_packet\_reg$ is used to record whether the packet is the first packet of a flow, and $Timestamp\_reg$ records the last packet's arrival time of a flow. In the meantime, three flag variables, i.e., IAT_flag, $1st\_flag$ and TCP_flag, are used to determine whether the memory space need to be initialised by three conditions: (i) exceeding IAT threshold, (ii) arrival of the first packet of a flow and (iii) receiving TCP termination flags.

The process of memory management in the pipeline is shown in Figure 4. A packet attached with TCP termination flags can trigger memory initialisation, setting the variable TCP_flag to 1. If it directly initialises memory space, the termination packet will be the first packet of a new flow hashed in the same memory space, i.e., the next packet will be the real first packet. Thus, *pHeavy* uses $First\_packet\_reg$ to record whether the packet is the first packet after processing TCP termination packets. In the $RegisterAction$ of $First\_packet\_reg$, the TCP termination packet (TCP_flag = 1) will set the value of the $Register$ to 0. Then the next packet (i.e., the first packet of a flow) fetches the value of $First\_packet\_reg$ (now equals 0) to initialise memory space while setting the value of $First\_packet\_reg$ to 1 for updating subsequent packets. Afterwards, in the $RegisterAction$ of $Timestamp\_reg$, a packet's arrival time (e.g., ingress timestamp) will be used to judge whether it triggers memory initialisation. If the IAT threshold has been exceeded, IAT_flag will be set to 1 while updating the value of $Register$, and the packet is the first packet of a new flow on this condition. Otherwise, only update the value of $Register$. In the end of the process, values of these three flags will determine whether to perform feature updating or memory initialising operations.

Table V
TOP THREE IMPORTANT FEATURES IN DECISION TREES

UNI1			UNI2	
# of packets	bmv2	Tofino (TCP)	# of packets	bmv2
6th	IAT_max dport IAT_total	PSH SYN len_max	5th	dport IAT_max len_min
8th/9th	len_avg dport ACK	ACK len_total SYN	7th	len_max len_avg len_total
14th	SYN dport ACK	ACK dport SYN	14th	IAT_total len_min IAT_avg
20th	ACK PSH len_avg	dport ACK	20th	IAT_avg IAT_total len_max

VII. EVALUATION

In this section, we evaluate the performance of *pHeavy* in both software switch (i.e., bmv2) and P4 hardware switch (i.e., Barefoot Tofino).

A. Performance metric and comparison schemes

F-measure: Since the change of occupation rate may not effect TPR and TNR, these two metrics can not correctly reflect the performance of classifiers in imbalanced dataset. F-measure ($\beta = 1$) is a metric widely used to evaluate binary classification. F-measure metric combines precision and recall as an effectiveness measure of classification, defined as:

$$F-measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

F-measure has more insights into the functionality of a classifier. It is very sensitive to the rate of imbalance of dataset [28]. The value of F-measure will be attenuated severely by more skewed distributions. Therefore, to evaluate the performance of *pHeavy* in extremely imbalanced distribution, F-measure

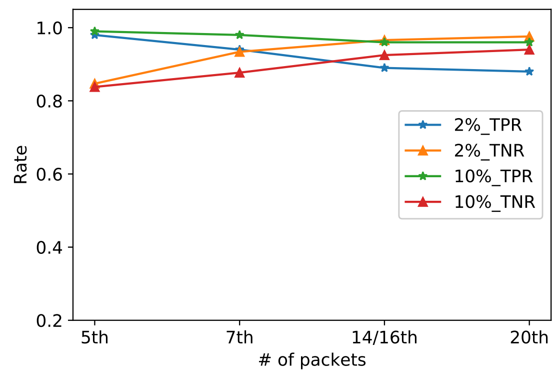


Figure 5. *pHeavy* selects four decision trees in different locations for prediction when α = 10% and α = 2% respectively

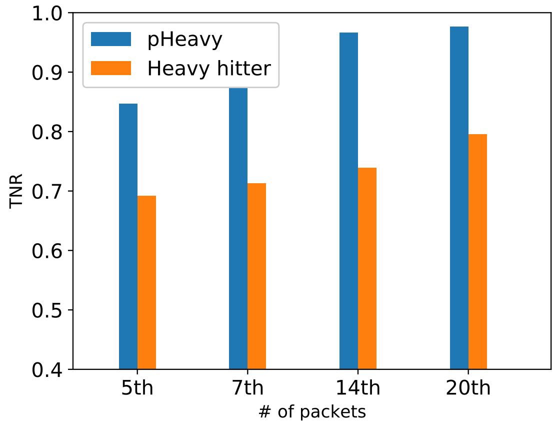


Figure 6. Comparison of TNR with heavy hitter

can be used to evaluate the same dataset with two occupation rates of heavy flow (α = 2% and α = 10%).

Heavy hitter: Heavy hitter [36] is a common method based on counters for heavy flow detection in the data plane. It can be easily integrated into switches but has a long detection time. Thus, we compare the TNR of *pHeavy* and heavy hitter after receiving few packets of each flow.

Machine learning in the controller: APPR [25] is a machine learning scheme for predicting network traffic. Previous work has run APPR algorithm in the controller to identify heavy flows [26] since it has high accuracy of prediction by only few packets of a flow. Considering the imbalanced data problem, random undersampling is performed on training data to keep the same number of minority and majority when training models by APPR.

B. Performance in P4 software switch

Accuracy: Figure 5 shows the training performance of *pHeavy* on both TPR and TNR. The experiment is conducted on UNI2 dataset. *pHeavy* arranges four decision trees for prediction in different locations, i.e., the prediction happens in the 5/7/14/20*th* packet when α = 2% and the 5/7/16/20*th* packet when α = 10% respectively. The result shows that *pHeavy* can keep high TPR while obtain high TNR. TPR decreases while TNR increases when a flow receives more packets. The reason is that each decision tree aims to keep high TPR while increasing TNR as high as possible. The first decision tree has a high TNR, partially contributed by correctly identifying flows not exceeding four packets, and these flows can be erased by the memory management subsequently.

Comparison with heavy hitter: Figure 6 shows the speed of prediction for *pHeavy* and heavy hitter. As we can see from this figure, heavy hitter is able to achieve 70% TNR because of the existence of short-lived flows. The TNR for both schemes grows steadily as more packets enter the switch, but our *pHeavy* consistently has more than 15% better performance.

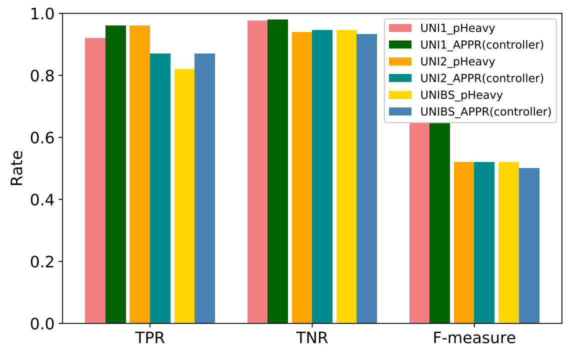
Comparison with machine learning in controller: We next compare the performance of *pHeavy* and APPR. This experiment evaluates three metrics, including TPR, TNR and F-measure, and is conducted on the three datasets (UNI1, UNI2 and UNIBS) with two occupation rates (α = 10% and α = 2%). Results in Figure 7 shows that *pHeavy* has comparable performance with APPR. First, both *pHeavy* and the APPR algorithm have high TPR and TNR. Second, *pHeavy* has good performance in F-measure when comparing with APPR. The value of F-measure is attenuated in *pHeavy* and APPR when the occupation rate of heavy flow declines. It is worth noting that *pHeavy* has a higher F-measure value when α = 2%. This is because the flow memory management of *pHeavy* filters out a portion of flows that meet the termination conditions before triggering the prediction of the decision tree. However, the core idea of APPR is defining "application layer round", which largely extends the number of available features to increase accuracy. Thus, APPR is inappropriate to be implemented in the data plane as it requires so many features and produces complicated machine learning models.

C. Performance in the P4 hardware switch

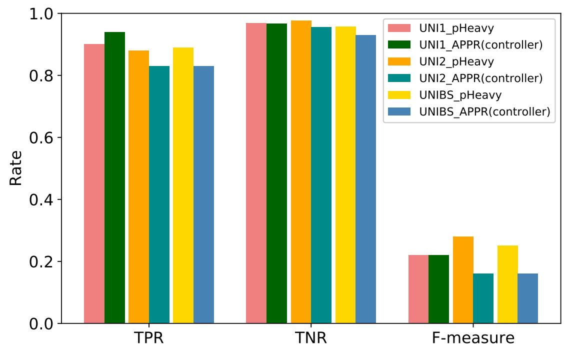
Our testbed for evaluating *pHeavy*'s performance in P4 hardware switch consists of two servers (equipped with 8 Intel Core i7-4771 CPU @ 3.50GHz), each configured with a 10Gbps NIC, and a P4 hardware switch (Flnet S9180-32X with Barefoot Tofino) connecting the servers. One server is used to replay real TCP network traffic (i.e., UNI1 and UNIBS) via Linux network traffic tool *Tcpreplay* [8], and the other one is responsible to receive the replayed packets.

Predicting at line rate: To evaluate the throughput of *pHeavy*, a simple switching application named *basic_switch* [4] is implemented for baseline comparison. *iperf* is used to measure throughput between the two servers. As we can see in Figure 8(a), *pHeavy* achieves an average throughput of 9.412Gbps, which is only 0.01% slower than 9.413Gbps achieved by the *basic_switch*. This clearly demonstrates that *pHeavy* is able to work at line rate.

Flow Prediction Time: Flow prediction time refers to the time interval between receiving the first packet of a flow to predicting it as a heavy flow or a non-heavy flow. Flows that do not trigger prediction (e.g., the number of packet is less than 5) are not included. We also implement heavy hitter which maintains a counter for each flow and set its threshold to the

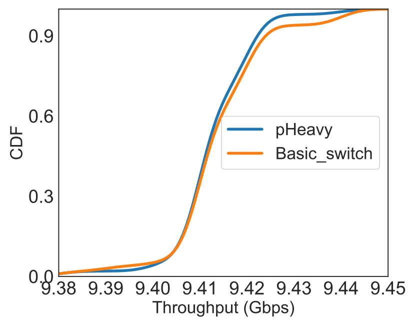


(a) 10% of heavy flow

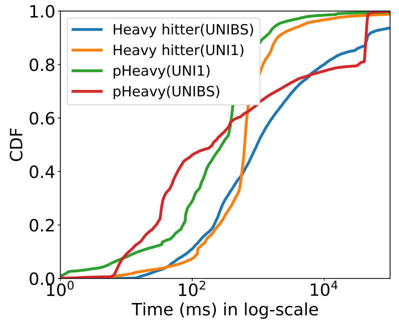


(b) 2% of heavy flow

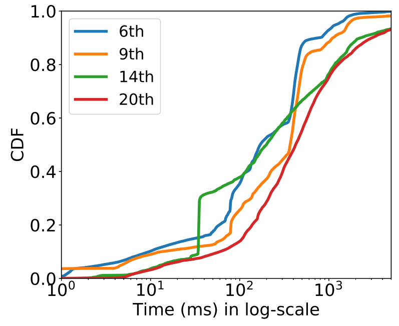
Figure 7. Comparison of *pHeavy* and APPR algorithm by three metrics



(a) Throughput of *pHeavy* and a simple use case in P4 hardware switches



(b) Comparison of pHeavy and heavy hitter in flow predicting time



(c) Flow prediction time at different locations (UNI1)

Figure 8. Performance on the P4 hardware switch

number that is the same as the position of the last decision tree (i.e., 20th) in *pHeavy* for comparison. Figure 8(b) shows the CDF of flow predicting time of both *pHeavy* and heavy hitter with UNI1 and UNIBS datasets. Evaluation results show that about 90% of flows can be predicted within 1.2s in UNI1 dataset. And the average flow prediction time is 2.6s and 10.3s for UNI1 and UNIBS respectively. In comparison, the time for heavy hitter is 7.9s and 31.8s respectively. Thus, *pHeavy*'s prediction time is 3x faster than heavy hitter on average. In the UNIBS dataset, flow predicting time of *pHeavy* increases because the dataset consists of many SSH tunnel flows that only transfer dozens of packets over tens of seconds. It is noted that heavy hitter has very low TNR (e.g., about 80% TNR in 20th packets, as shown in Figure 6) when it has the same predicting location with *pHeavy*.

In addition, Figure 8(c) shows flow predicting time in different predicting locations (i.e., 6th, 9th, 14th and 20th). As expected, the earlier the location of prediction, the shorter the flow predicting time. Flow predicting times of over 91% at the 6th and 9th packets, 84.7% at the 14th packets, and 75.4% at the 20th packets are completed within 1s.

D. Optimization for the programmable data plane

Selection of decision trees: *pHeavy* adopts a greedy algorithm to select decision trees. To show the effectiveness of the selection algorithm, a random method is also designed for comparison. As shown in Figure 9, although the two methods have the same F-measure value, the greedy algorithm has a higher TPR while smaller accumulated size of decision trees. Thus, the greedy algorithm can effectively select decision trees to be implemented in the data plane.

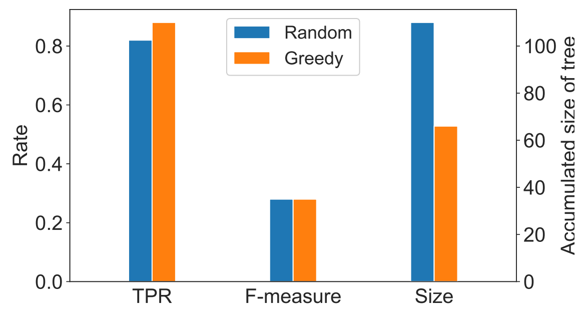


Figure 9. Comparison of random algorithm and greedy algorithm for decision tree selection

Optimization on the size of decision trees: Figure 10 shows the performance of the training algorithm for both *pHeavy* and APPR in terms of the size of decision trees (e.g.,

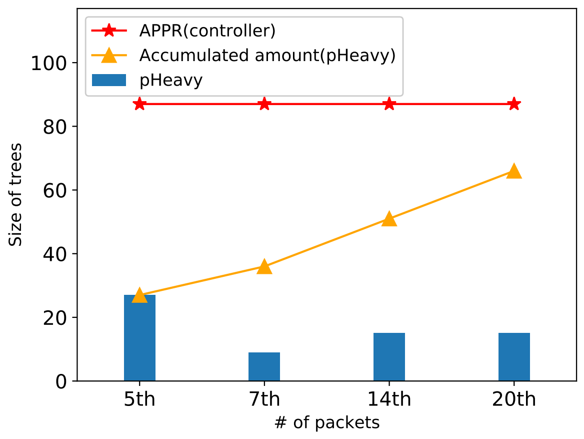


Figure 10. Comparison of the size of decision trees with APPR algorithm

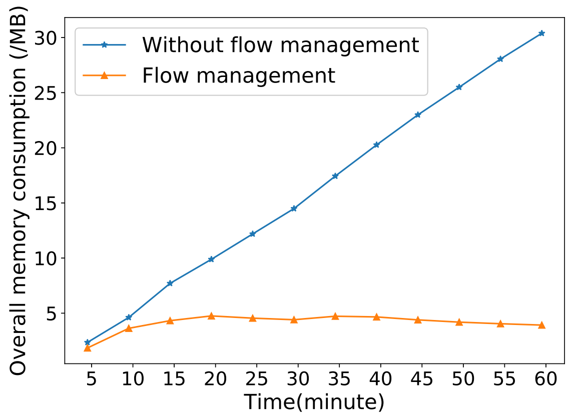


Figure 11. The overall memory consumption in different time conducted by UNI2 dataset

amounts of nodes). Decision trees produced by *pHeavy* are smaller by 5.4x on average than that of APPR. Although the accumulated size in *pHeavy* is close to that of APPR, *pHeavy* shares computation of prediction in a flow with several smaller predictions in different packet locations, which effectively enables it to avoid effects of processing per packet in the data plane. Thus, *pHeavy* improves the feasibility of implementing decision trees in the programmable data plane.

Optimization on the amount of stored flows: Since the scarce memory in the programmable data plane limits the storage of flows, we also study the number of flows that *pHeavy* can concurrently classify by giving a certain memory. Similar to [15], the number of concurrent flows is estimated according to the given memory divided by total bits of information to be stored per-flow. Storing all mentioned features of each flow, *pHeavy* can handle about 200 thousands of concurrent flows per 10MB memory. Furthermore, Figure 11 shows overall memory consumption that *pHeavy* needs to provide in different time spots during experiments. *pHeavy* only requires 5MB memory in the P4 software switch to store all features of up to 100k active flows. Thus, *pHeavy* is able to schedule the memory usage effectively and dynamically.

VIII. Conclusion

In this paper, we present *pHeavy* which predicts heavy flow via machine learning algorithm in the programmable data plane. *pHeavy* consists of two phases, offline model training and online inference. In the first phase, *pHeavy* proposes a training algorithm to tackle the imbalanced data problem while minimizes the size of trees to implement in the data plane. In the second phase, *pHeavy* provides memory management to increase the amount of concurrent flow, and replaces unsupported operations with approximate values. Experiment results show that *pHeavy* can effectively predict heavy flow in early stages with high accuracy at line rate.

There is an open problem in *pHeavy* that is worth exploring further. The data plane is hard to install machine learning models that are trained by large dataset (e.g., several days network traffic), due to the large size of decision trees. A solution is to split large traffic into several parts based on periods of time, and the controller (e.g., P4 runtime [6]) can dynamically configure and modify machine learning models.

Acknowledgment

This work has been partially supported by Chinese National Research Fund (NSFC) No. 61772235 and 61872239; Natural Science Foundation of Guangdong Province No. 2020A1515010771; Science and Technology Program of Guangzhou No. 202002030372; The UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/2 and EP/P004024/1; InnovateUK grant 106199-47198; Guangdong Key Lab of AI and Multi-modal Data Processing; BNU-UIC Institute of Artificial Intelligence and Future Networks funded by Beijing Normal University (Zhuhai) and AI-DS Research Hub.

References

[1] Data set for IMC 2010 data center measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html. Accessed on: June 20, 2021.
[2] dpkt 1.9.2 documentation. https://dpkt.readthedocs.io/en/latest/. Accessed on: June 20, 2021.
[3] imblearn.under_sampling.randomundersampler. https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.under_sampling.RandomUnderSampler.html. Accessed on: June 20, 2021.
[4] Intel tofino series programmable ethernet switch asic. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html. Accessed on: June 20, 2021.
[5] p4lang/behavioral-model. https://github.com/p4lang/behavioral-model. Accessed on: June 20, 2021.
[6] p4lang/pi: An implementation framework for a p4runtime server. https://github.com/p4lang. Accessed on: June 20, 2021.
[7] scikit-learn machine learning in python. https://scikit-learn.org. Accessed on: June 20, 2021.
[8] Tcpreplay - pcap editing and replaying utilities. https://tcpreplay.appneta.com/. Accessed on: June 20, 2021.
[9] Unibs: Data sharing. http://netweb.ing.unibs.it/~ntw/tools/traces/. Accessed on: June 20, 2021.
[10] Weka the workbench for machine learning. https://www.cs.waikato.ac.nz/ml/weka/. Accessed on: June 20, 2021.
[11] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 89–92, 2010.

[12] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking*, 2020.

[13] Ran Ben Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Optimal elephant flow detection. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[14] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, pages 1–12, 2011.

[15] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pforest: In-network inference with random forests. *arXiv preprint arXiv:1909.05680*, 2019.

[16] Shou-Chieh Chao, Kate Ching-Ju Lin, and Ming-Syan Chen. Flow classification for software-defined data centers using stream mining. *IEEE Transactions on Services Computing*, 12(1):105–116, 2016.

[17] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 254–265, 2011.

[18] Pedro Domingos. Metacost: A general method for making classifiers cost-sensitive. In *Proceedings of the ACM SIGKDD conference*, pages 155–164, 1999.

[19] Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Quantifying the accuracy of the ground truth associated with internet traffic traces. *Computer Networks*, 55(5):1158–1167, 2011.

[20] Cristian Estan, Ken Keys, David Moore, and George Varghese. Building a better netflow. *ACM SIGCOMM Computer Communication Review*, 34(4):245–256, 2004.

[21] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 339–350, 2010.

[22] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational ip networks: Methodology and experience. *IEEE/ACM Transactions On Networking*, 9(3):265–279, 2001.

[23] Francesco Gringoli, Luca Salgarelli, Maurizio Dusi, Niccolo Cascarano, Fulvio Risso, and KC Claffy. Gt: picking up the truth from the ground for internet traffic. *ACM SIGCOMM Computer Communication Review*, 39(5):12–18, 2009.

[24] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

[25] Nen-Fu Huang, Gin-Yuan Jai, Han-Chieh Chao, Yih-Jou Tzang, and Hong-Yi Chang. Application traffic classification at the early stage by characterizing application rounds. *Information Sciences*, 232:130–142, 2013.

[26] Yuan-Hao Huang, Wen-Yueh Shih, and Jiun-Long Huang. A classification-based elephant flow detection method using application round on sdn environments. In *19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 231–234. IEEE, 2017.

[27] Nathalie Japkowicz. Assessment metrics for imbalanced learning. *Imbalanced learning: Foundations, algorithms, and applications*, pages 187–206, 2013.

[28] László A Jeni, Jeffrey F Cohn, and Fernando De La Torre. Facing imbalanced data–recommendations for the use of performance metrics. In *2013 Humaine association conference on affective computing and intelligent interaction*, pages 245–251. IEEE, 2013.

[29] Miroslav Kubat, Stan Matwin, et al. Addressing the curse of imbalanced training sets: one-sided selection. In *ICML*, volume 97, pages 179–186. Nashville, USA, 1997.

[30] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 201–206, 2004.

[31] Victoria López, Alberto Fernández, and Francisco Herrera. On the importance of the validation technique for classification with imbalanced datasets: Addressing covariate shift when data is skewed. *Information Sciences*, 257:1–13, 2014.

[32] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE communications surveys & tutorials*, 10(4):56–76, 2008.

[33] Pascal Poupart, Zhitang Chen, Priyank Jaini, Fred Fung, Hengky Susanto, Yanhui Geng, Li Chen, Kai Chen, and Hao Jin. Online flow size prediction for improved network routing. In *IEEE 24th International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE, 2016.

[34] Konstantinos Psounis, Arpita Ghosh, Balaji Prabhakar, and Gang Wang. Sift: A simple algorithm for tracking elephant flows, and taking advantage of power laws. In *43rd Allerton Conference on Communication, Control and Computing*, 2005.

[35] J Ross Quinlan. *C4.5: programs for machine learning*. Elsevier, 2014.

[36] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017.

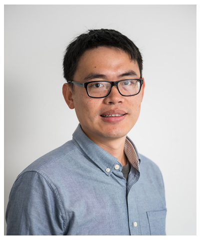
[37] Binfeng Wang and Jinshu Su. A survey of elephant flow detection in SDN. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–6. IEEE, 2018.

[38] Peng Xiao, Wenyu Qu, Heng Qi, Yujie Xu, and Zhiyang Li. An efficient elephant flow detection with cost-sensitive in SDN. In *2015 1st International Conference on Industrial Networks and Intelligent Systems (INISCom)*, pages 24–28. IEEE, 2015.

[39] Xiaoquan Zhang, Lin Cui, Kaimin Wei, Fung Po Tso, Yangyang Ji, and Weijia Jia. A survey on stateful data plane in software defined networks. *Computer Networks*, 184:107597, 2021.

Xiaoquan Zhang Xiaoquan Zhang, born in 1993. Currently he is a postgraduate student in Jinan University. He received the B.E. degree in computer science and technology from Chengdu University of Technology, China, in 2016. His current research interests includes stateful data plane/programmable data plane, software-defined networking, and machine learning in computer network.

Lin Cui is currently a professor in the Department of Computer Science at Jinan University, Guangzhou, China. He received the Ph.D. degree from City University of Hong Kong in 2013. He has broad interests in networking systems, with focuses on cloud data center networking, software defined networking (SDN), NFV, programmable networking, distributed systems and so on.



Fung Po Tso received his B.Eng., M.Phil. and Ph.D. degrees from City University of Hong Kong in 2006, 2007 and 2011 respectively. He is currently a senior lecturer in the Department of Computer Science at the Loughborough University. His research interests include: network policy management, network measurement and optimisation, service chaining, data centre networking, software defined networking (SDN), and edge computing.

Weijia Jia is currently a Chair Professor and Director of BNU-UIC Institute of Artificial Intelligence and future Networks, VP for Research of BNU-HKBU United International College. His research interests include smart city, IoT, knowledge graph constructions, multicast and anycast QoS routing protocols, wireless sensor networks, and distributed systems. He has over 500 publications in prestigious international journals/conferences and research books and book chapters.