RESEARCH ARTICLE OPEN ACCESS

Real-Time Flow-Level Heavy Hitter Prediction on P4 Programmable Switches

WeiJian Li*, Lin Cui**
*(Department of Computer Science, Jinan University, Guangzhou, China
Email: weijianli@stu2022.jnu.edu.cn)
** (Department of Computer Science, Jinan University, Guangzhou, China
Email: tcuilin@jnu.edu.cn)

----------**********----------

Abstract:

Identifying heavy hitters is essential for maintaining high network performance and security. Heavy hitters can signal potential issues such as network congestion, cyberattacks, or misconfigurations. Meanwhile, P4 programmable switches offer a promising solution for offloading heavy hitter detection to the data plane. Nevertheless, current methods are often limited by their dependence on control plane decisions, which introduce latency and overhead or lead to inefficient predictions and suboptimal real-time performance due to the data plane's constrained computational capability. This paper introduces the TimeSlot Predictor (TSP), a novel real-time flow-level prediction method that employs decision tree models on P4 programmable switches to address these challenges. TSP divides network flows into discrete time slots for real-time analysis, utilizing temporal data to forecast future heavy hitter trends and predict heavy hitters within current time slots. This approach enhances prediction accuracy and adapts to dynamic traffic patterns. We have implemented TSP on both P4 hardware switches (with *Intel Tofino* ASIC) and software switches (*BMv2*). Extensive experiments demonstrate that TSP predicts 98% of heavy hitters within an average time of 6 seconds.

Keywords — **Heavy Hitter, P4 Programmable Switch, Real-Time, Decision Tree.**

----------**********----------

I. INTRODUCTION

In high-performance networks, efficiently identifying heavy hitters is essential for maintaining both performance and security [1]. Heavy hitters, which are flows consuming a significant portion of network resources, are crucial for applications such as traffic engineering [2], anomaly detection [3], and network management [4]. Traditionally, heavy hitter detection is implemented on servers [5], [6] with general-purpose CPU, which requires sampling from switches, incurring communication overhead and delay.

The rapid development of programmable data planes, especially P4 programmable switches, offers new opportunities for heavy hitter detection [7]. Programmable data planes leverage the coordination of multiple stages of Match Action Units (MAUs) deployed in pipelines, enabling programmability of the pipeline in the data plane. Utilizing the ASIC chips in the programmable data plane, P4 programmable switches can achieve line-rate throughput (e.g., 6.4 Tbps in *Intel Tofino*) [8]. Offloading the heavy hitter logic to the programmable data plane can reduce communication overhead between the data plane and the control plane (server), saving the CPU load on servers. There are many works focus on heavy hitter detection on the data plane, such as Hedera [9], Helios [10], and EFD [11]. To identify heavy hitters more timely, so operators can respond

TABLE I

Comparison of different works for heavy hitter.

Works	Methods	Category	Entirely on Data Plane	Slot-based (Real-Time)	Lightweight
APPR [3]	Sampling	Detection	×	×	-
NetFlow [5]	Sampling	Detection	×	×	-
HG-LDP [6]	Sampling	Detection	×	×	-
Hedera [9]	Sampling	Detection	×	×	-
Helios [10]	Sampling	Detection	×	×	-
Flowseer [12]	Decision Tree	Prediction	×	×	-
EFD [11]	Decision Tree	Prediction	×	×	-
Hashpipe [15]	Top-k	Detection	✓	×	✓
Devoflow [16]	Top-k	Detection	✓	×	✓
OEFD [17]	Top-k	Detection	✓	×	✓
pHeavy [13]	Decision Tree	Prediction	✓	×	×
pForest [14]	Decision Tree	Prediction	✓	×	×
TSP(ours)	Decision Tree	Prediction	✓	✓	✓

quickly, heavy hitter prediction in the data plane is necessary.

However, achieving effective heavy hitter prediction in the data plane is challenging. First, machine learning models for prediction are constrained by limited computing capability (e.g., do not support floating-point and multiply-divide calculations) and memory resources (e.g., ~ 120Mbits SRAM per pipeline for Tofino ASIC). Additionally, the data plane pipeline is designed with a limited number of stages (e.g., 12 stages in Tofino ASIC), and its inability to support logical operations such as loops further complicates the development of efficient prediction algorithms. Second, the dynamics of network traffic flows make it more complex and difficult to perform effective heavy hitter prediction, which requires the ability to adapt to these dynamics and make decisions in real time.

Existing solutions either need the assistance of control plane or fail to adapt to the dynamics. For example, some methods [11], [12] deploy decision trees in the control plane to predict heavy hitters based on information collected by the programmable data plane, incurring communication overhead and latency between the control plane and the data plane, resulting in delayed prediction results. Some solutions deploy pre-trained decision tree models entirely in the data plane e.g., pHeavy [13] and pForest [14]. These methods can achieve high throughput and prediction accuracy, but without considering the temporal changes of flows making them insensitive to transient network changes and challenging to predict in real-time in dynamic network environments.

To address the above challenges, this paper proposes a time slot-based heavy hitter prediction method, named TimeSlot Predictor (TSP). TSP leverages decision tree machine learning models to predict heavy hitters in real-time, directly on both P4 hardware switches (with *Intel Tofino* ASIC) and software switches (*BMv2*). By segmenting network traffic data into discrete time slots and only recording the most recent data for making decisions, TSP performs effective flow-level prediction adapting to the network dynamics. Optimizing feature extraction and model efficiency is crucial to operate within these constraints while maintaining high prediction accuracy and processing speed. To summarize, we mainly make the following contributions:

1) **Real-time heavy hitter prediction on data plane**: We proposed a time slot-based heavy hitter prediction method, called TSP, which performs real-time prediction on P4 switches using decision tree machine learning models. By segmenting network traffic data and applying these models, TSP improves prediction accuracy and efficiency.
2) **Optimized feature extraction and model design**: To handle pipeline constraints of P4 programmable switches, we proposed an optimized feature extraction scheme and training efficient machine learning models that

operate on data plane, ensuring high performance under limited resources.

3) **Experimental validation and performance evaluation**: We have implemented TSP on both P4 hardware switches (with *Intel Tofino* ASIC) and software switches (*BMv2*). Extensive evaluations demonstrate the effectiveness and performance of TSP. Results show that TSP predicts 98% of heavy hitters within an average time of 6 seconds.

The rest of the paper is organized as follows: Section II discusses related works and motivations for identifying heavy hitters on data plane. Section III presents the system design of TSP, including the training phase in the control plane and the inference phase in the data plane. Section IV evaluates the proposed methods and provides performance results under different system configurations. Finally, Section V concludes the paper.

II. RELATED WORKS

The identification of heavy flows in network traffic is crucial for ensuring network performance and security. APPR [3], NetFlow [5] and HG-LDP [6] are examples of server-based approaches that leverage the general-purpose computing power of CPUs to implement complex logic for effective heavy hitter detection. While these methods can be highly effective, they often introduce significant CPU load. Additionally, the bloated software protocol stack makes it challenging to meet the high throughput demands of high-performance networks. To address these issues, recent research has turned its attention to programmable data planes. As shown in Table I, existing solutions implemented in programmable data planes can be categorized into two categories: detection, and prediction.

A. Heavy Hitter Detection

Many methods estimate the presence of heavy hitters by analyzing random traffic samples with the assistance of the control plane, e.g., Hedera [9] and Helios [10]. Such approach inevitably introduces communication overhead between the control plane and the data plane. Works like Hashpipe [15], Devoflow [16], and OEFD [17] implement P4-based algorithms that use a multi-stage pipeline to track the top k heavy flows with high accuracy. These approaches can quickly process large volumes of data with minimal computational resources. However, they often face challenges in balancing detection accuracy and resource consumption. Their reliance on predefined thresholds and packet counters can make them less adaptable to dynamic network conditions and limit their ability to predict future trends. And it is difficult to predict and prevent heavy hitters that will occur in the future.

B. Heavy Hitter Prediction

Some works deploy Machine Learning-based techniques on the control plane. FlowSeer [12] employs a two-stage prediction algorithm, utilizing decision trees and Hoeffding trees for improved accuracy. Similarly, EFD [11] deploys decision trees on the control plane to predict heavy hitters based on information collected from the data plane. These methods leverage the high throughput and low latency characteristics of the data plane to meet the demands of high-performance networks. However, these methods rely on the control plane for detection, which still incurs communication overhead and latency between the control plane and the data plane, resulting in delayed prediction results.

Some works implement efficient prediction with Machine Learning-based techniques entirely on the data plane. pHeavy [13] uses decision tree models directly implemented on the programmable data plane for packet-level heavy hitter prediction. pForest [14] leverages random forest models for network flow prediction, optimized for the computational constraints of P4 switches. While these methods address some issues, they often lack the capability for temporal predictions, limiting their effectiveness in dynamic network environments. They primarily focus on packet-level prediction and involve complex operations that do not fully exploit the temporal dynamics of network

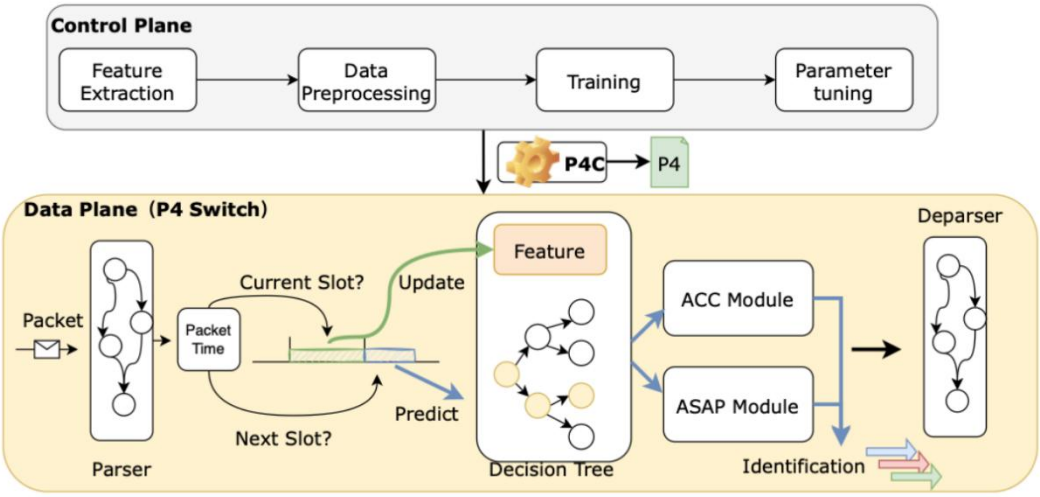


Fig. 1 TSP Pipeline Overview.

traffic. Compared to top-k methods, these approaches require storing more flow information, which consumes more memory resources and can be less lightweight. Moreover, the reliance on global data can delay the quick identification of sudden traffic bursts due to the dominant influence of historical data.

Overall, while significant progress has been made in the field of heavy hitter prediction, challenges remain in achieving real-time, accurate, and resource-efficient prediction, especially in highly dynamic network environments.

III. SYSTEM DESIGN

A. Overview

As shown in Figure 1, the system design of TSP comprises the control plane and the data plane. The control plane is responsible for model training, utilizing a decision tree classifier with traffic features and addressing data imbalance with One-sided Selection and Random Under-Sampler. It processes the dataset through four stages: Feature Extraction, Data Preprocessing, Training, and Parameter Tuning, resulting in a deployable decision tree model. This model's corresponding p4 code is compiled and deployed on the data plane via P4C.

In the data plane, TSP performs real-time heavy hitter prediction by recording flow information as packets traverse the pipeline and periodically executing the decision tree based on *slot_size*. The decision tree model ensures a balance between accuracy and latency by applying ACC Module and ASAP Module flexibly.

B. Feature Extraction

*1) **Feature Selection:*** To facilitate effective heavy hitter prediction, TSP selects network traffic features that are feasible to implement within the constraints of a programmable data plane [18]. Table II enumerates these features, including various metrics related to packet inter-arrival time, packet length, and TCP flags.

TABLE II
Network Traffic Features

Name	Description
IAT	max Maximum packet inter-arrival time of the flow
IAT	min Minimum packet inter-arrival time of the flow
IAT	mean Average packet inter-arrival time of the flow
Length	max Maximum packet length of the flow
Length	min Minimum packet length of the flow
Length	mean Average packet length of the flow
Length	total Total packet length of the flow
ACK flag	ACK flag counter of the flow
FIN flag	FIN flag counter of the flow
SYN flag	SYN flag counter of the flow
PSH flag	PSH flag counter of the flow
RST flag	RST flag counter of the flow
ECE flag	ECE flag counter of the flow
Packet count	Packet counter of the flow

2) ***Feature Extraction Procedure:*** TSP defines the flows of each time slot that exceed a defined occupancy threshold (α) as heavy hitters. It classifies traffic features within a time slot into two categories, i.e., *Positive Samples* and *Negative Samples*, depending on whether they are related to heavy hitters or not.

TSP employs the *dpkt* library to parse network traffic and extract flow features, storing them as DataFrames for each time slot. The feature extraction process is illustrated in Figure 2.

In *Slot 1*, Flow C is identified as a heavy hitter, with its features collected as positive samples, while features of Flow A are collected as negative samples. In *Slot 2*, the features of Flow C continue to be collected as positive samples, and both Flow A and Flow B are marked as non-heavy hitters. By *Slot 3*, Flow A is upgraded to a heavy hitter, and its features are collected as positive samples.

C. Data Preprocessing

Before training, achieving a balanced dataset is necessary. An imbalanced dataset makes the model biased towards the majority class, which can result in poor performance on the minority class. Additionally, this imbalance presents challenges for training effective models without exceeding the memory limitations of P4 programmable switches.

As shown in Figure 3, in different *slot_size* and occupancy threshold (α), the initial sample distributions revealed a severe class imbalance, with an average imbalance ratio of 0.015 (ranging from 0.0037 to 0.045).

To address this, TSP performs feature filtering by removing flow features with fewer than 20 packets, as these flows generally lack sufficient data and have high information entropy, leading to misclassification. After filtering, 10% of positive samples and 90% of negative samples were removed, reducing the class imbalance to an average ratio of 0.126 (ranging from 0.033 to 0.264).

Despite this improvement, the data imbalance persisted, necessitating further balancing using One-Sided Selection (OSS) and random under-

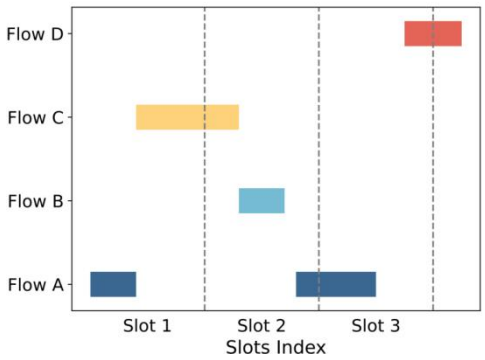


Fig. 2 Example of slotted-based feature extraction.

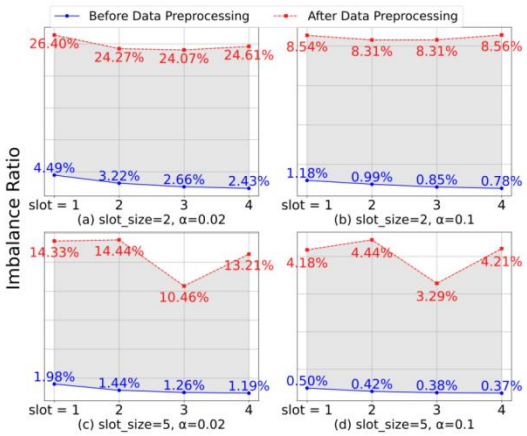


Fig. 3 Comparison of Sample Distribution before and after feature filtering.

sampling to create a more balanced training set. As shown in Algorithm 1, OSS leverages Tomek Links [19] to eliminate redundant and noisy negative samples, thereby preserving representative samples and resulting in a balanced training set. Tomek Links are pairs of samples, e.g., (x, y), where each sample has a different class label. If no sample z exists such that the distance between x and z or y and z is less than the distance between x and y, then (x, y) is identified as a Tomek Link. The use of Tomek Links is crucial in OSS for refining the training dataset by specifically targeting and removing borderline or noisy samples. This process ensures that the training set is balanced and clean, improving the model's performance by reducing the likelihood of misclassifications caused by noisy data.

Algorithm 1 One-Sided Selection

Require: S: original training set, C: positive samples from S and one randomly selected negative sample
Ensure: T: balanced training set

```
Initialize C with all positive samples and one negative sample from S
for each sample in S do
Classify S using 1-NN rule with samples in C
Move misclassified samples to C
end for
while all positive samples are not retained do
    for each sample in C do
        Remove negative samples participating in Tomek Links
    end for
end while
```

D. Prediction Model Design

1) ***Prediction Model Selection:*** TSP employs decision trees as the predictive model due to their straightforward structure and operational simplicity, making them suitable for deployment in programmable switches [20]. Despite addressing data imbalance in the Data Processing stage, some imbalance persists, challenging the performance of decision trees. These models, while versatile, can overfit in the presence of highly imbalanced data, where negative samples vastly outnumber positive ones.

Pruning decision trees can enhance generalization by eliminating leaves with low-class probability estimates. However, this technique might inadvertently remove branches that capture minor yet critical concepts in imbalanced datasets, thus not addressing the core issue. In scenarios with rare positive samples, decision trees often misclassify mixed sample regions as negative due to the dominance of negative samples. To mitigate this, TSP employs cost-sensitive decision trees, which handle imbalanced data by assigning different misclassification costs to different classes. In this context, higher costs are assigned to misclassifying positive samples, thereby biasing the model to be more cautious about these critical samples. This approach helps in improving the classification performance of the minority class without overly simplifying the decision tree.

2) ***Prediction Model Implementation:*** TSP utilizes the decision tree classifier with an optimized Classification and Regression Tree (CART) algorithm [21]. This method uses Gini impurity instead of information entropy, reducing computational overhead by avoiding logarithmic operations. CART identifies splits that minimize the Gini impurity for each feature and threshold, weighted by node sizes, according to the following loss function Equation (1):

$$J(k, t_k) = \sum \frac{S_i}{S} P_i, (1)$$

where S_i is the number of instances in subset i and P_i is the impurity of subset i.

Given the NP-complete nature of finding the optimal tree, CART employs a greedy algorithm that stops splitting when a predefined maximum depth (*max_depth*) is reached or when further splits do not significantly reduce impurity.

To handle data imbalance, TSP leverages sample weight and class weight parameters. Sample weight adjusts the weights of individual samples, while class weight adjusts the weights of each class. Using class weight = balanced and the default sample weight yielded the best predictive performance by equalizing class influence.

Hyperparameter tuning is performed using GridSearchCV, which automates the search for optimal hyperparameter combinations through exhaustive search and cross-validation, thereby enhancing model optimization. Additionally, TSP sets *min_samples_leaf* to 200 to ensure the construction of a locally optimal decision tree.

3) ***Model Regularization:*** While pruning does not resolve data imbalance, it is crucial for maintaining manageable tree sizes in decision tree models. TSP employs post-pruning to convert certain nodes into leaves, starting from the bottom of the tree. This process is validated by comparing accuracy before and after pruning. Specifically, TSP uses Cost Complexity Pruning (CCP), controlled by the parameter *ccp_alpha*. Higher *ccp_alpha* values result in more extensive pruning, whereas lower values preserve more of the original tree structure.

The steps for CCP involve calculating the *ccp_alpha* values for each node from the bottom up, pruning the tree accordingly, and selecting the optimal subtree based on cross-validation scores.

The detailed algorithm for CCP is shown in Algorithm 2.

To balance model complexity with data plane constraints, TSP adjusts *ccp_alpha*, typically favoring smaller values for practical implementation. The final predictive model is evaluated using True Positive Rate (TPR), True Negative Rate (TNR), and F1-Score, determined by averaging results from multiple training runs.

After optimizing the decision tree, TSP translates the model into P4 code using P4C, ensuring seamless deployment on programmable switches.

Algorithm 2 Optimal Subtree Selection Using CCP Apha

Require: T: complete decision tree, D: dataset, k: number of cross-validation folds
Ensure: T_{opt}: optimal subtree

1: Initialize lists A (*ccp_alpha* values), S (subtrees), V (validation scores)
2: **for** each node from bottom up **do**
3: Calculate and store *ccp_alpha* in A
4: **end for**
5: **for** each *ccp_alpha* in A **do**
6: Prune T to obtain subtree T_i and add T_i to S
7: **end for**
8: **for** each subtree T_i in S **do**
9: Initialize list F (fold scores)
10: **for** each fold f in k **do**
11: Split D into training and validation set: D_{train}, D_{val}
12: $score_f \leftarrow$ Train T_i on D_{train} and evaluate on D_{val}
13: Add $score_f$ to F
14: **end for**
15: Calculate and store average score $score_i$ in V
16: **end for**
17: Identify T_{opt} as subtree in S with maximum score in V
18: **return** T_{opt}

E. Data Plane Design

1) ***Pipeline Overview:*** The P4 language offers a robust framework for programming packet processing functions directly on data plane of network hardware. TSP, which is a packet processing application developed using P4, operates within this framework but faces significant challenges due to its packet-by-packet processing limitation. Specifically, TSP can only access information related to the current packet passing through the pipeline, making it difficult to predict features based on other flows simultaneously.

Balancing the accuracy and latency of flow predictions presents another challenge. The timing of feature collection is critical: in load-balancing scenarios, a few false negatives might be tolerable, prioritizing prediction speed over accuracy. Conversely, when identifying heavy flows suspected of denial-of-service (DoS) attacks, accuracy is paramount to avoid unnecessary alerts that could overwhelm network operators.

To address these challenges, TSP devised a stage-sequenced logic execution scheme, suitable for the constrained resources of hardware switch pipelines (As shown in Figure 1). Initially, packets are processed by the Parser, which extracts packet headers to compute flow features such as TCP flags and inter-arrival times (IAT). This information accompanies the packet to subsequent stages. Upon entering the pipeline (Ingress and Egress), packets are hashed into their respective flows based on five-tuple information (source/destination IP address, source/destination port, protocol number). At the end of the pipeline, all packets undergo deparsing before being reconstructed for transmission.

The P4 programmable switch provides ingress pipeline timestamps, enabling the calculation of time-dependent variables (e.g., IAT). All features are stored in registers and manipulated using basic P4-supported operations (addition, subtraction, hashing). For example, if the ACK flag of an incoming packet is set, the ACK flag counter increments accordingly. Since P4 does not support division operation, TSP employs an exponentially weighted moving average (EWMA) to perform averaging, defined as Equation (2):

$$E_t = \alpha \times x_t + (1 - \alpha) \times E_{t-1}, (2)$$

where x_t represents the counters at time t, and E_t represents the EWMA at time t. Setting α to 0.5 allows the average calculation to be implemented using bit-wise operations within the switch.

This approach enables TSP to efficiently manage the trade-offs between accuracy and latency, ensuring effective flow prediction within the constraints of the P4 programmable switch environment.

2) ***Slot Module:*** Given the packet-driven nature of P4 programmable data plane, TSP can only obtain

the hash value of the current packet's flow as it enters the pipeline. This hash value is used to predict features collected from the previous slot via a decision tree. However, this method cannot predict features for other flows based on the current packet, which is a significant limitation. To address this issue, TSP employs two registers to store time information: *next_time_reg* and *slot_time_reg*, ensuring that each flow in every slot enters the prediction stage.

The *next_time_reg* register records the end time of each flow for the current slot, while the *slot_time_reg* maintains the end time of the current slot, shared among all flows. When the first packet of a new slot enters the switch, TSP compares the current packet time (*packet_time*) with the next time value from *next_time_reg*. It then updates the necessary registers to ensure the correct collection of flow features.

For example, consider a *slot_size* of 5 seconds. When Flow A's first packet arrives, TSP compares *packet_time* with *next_time*, which is initialized to the first slot's end time of 5 seconds. If *packet_time* is less than *next_time* and it is the flow's first packet, *slot_time* is obtained from *slot_time_reg*, and next time is updated accordingly.

When the first packet of a new slot arrives, TSP updates slot time to the new slot's end time. This process ensures the correct feature collection for new flows appearing in the current slot. After updating the necessary values, TSP inputs the collected features into the prediction model. The prediction results then clear the feature registers, preparing for the next slot's feature collection.

*3) **Prediction Module:*** The prediction module begins with two essential checks. First, it discards features if the packet count (packet count) for the current flow in the previous slot is less than 20, ensuring data consistency with pre-training standards. Second, it reads the prediction flag (*flag_reg*) to determine if the flow has already been predicted (*flag* = 1) or still requires prediction (*flag* = 0).

The prediction process employs four decision trees, each predicting the flow status for the subsequent slot. The results are stored in *res_flag*, indicating the number of slots until the flow becomes a heavy hitter. Specifically, *res_flag* = 0 signifies that the flow will not become a heavy hitter in the next four slots.

A critical challenge in online classification is determining the optimal timing for predictions. TSP addresses this by fixing the timing to slot intervals, thereby shifting the challenge to the number of predictions required for accuracy. Premature decisions may lack confidence, rendering them unsuitable for high-accuracy scenarios such as DoS attack detection. Conversely, delayed predictions risk service disruption or resource wastage.

To address these issues, TSP offers two models: ACC and ASAP. The ACC model prioritizes accuracy, while the ASAP model focuses on rapid predictions. In the ASAP model, predictions stop once a flow is identified as a heavy hitter, updating the flow's flag to 1. The ACC model employs two additional registers, *cur_flow_reg* and *heavy_flow_reg*, to ensure accuracy by comparing consecutive results.

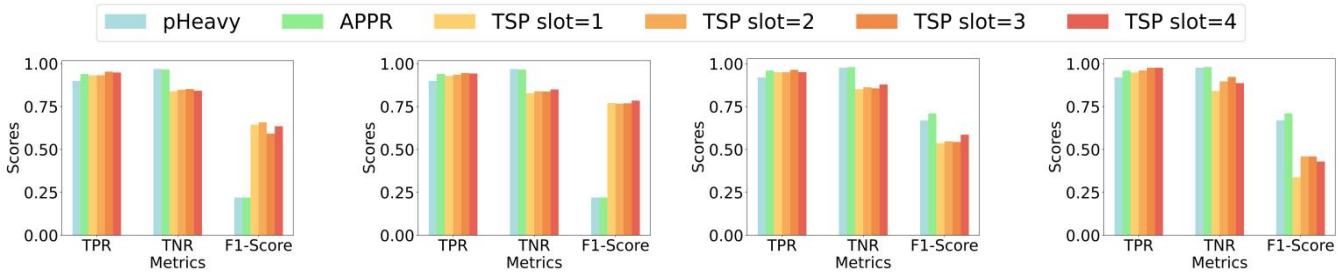
Upon receiving a termination signal (*flag* = 1), TSP releases the corresponding flow's memory in the programmable data plane, completing the prediction cycle.

This approach ensures that TSP can manage flow predictions efficiently within the programmable data plane, balancing accuracy and speed according to network requirements.

IV. EVALUATION

A. Experiment Settings

Testbed Setup. In the control plane, TSP utilizes the *dpkt* library to extract network traffic features. To address data imbalance, we apply One-Sided Selection and Random Under Sampling techniques. The dataset is split into training (70%) and test (30%) sets using *scikit-learn*. A decision tree classifier is then trained and selected as the prediction model. In the data plane, we simulate the network environment using *Mininet* [22], deploying TSP on *BMv2* [23] and *Intel Tofino* [24] switches. The trained model is converted to P4 code for real-time heavy hitter prediction. For validation, we use real network datasets from the UNI dataset [25], which includes packet traces from two university data centers, UNI1 and UNI2, with UNI1



(a) $\alpha = 2\%$, $slot\_size = 2$ (b) $\alpha = 2\%$, $slot\_size = 5$ (c) $\alpha = 10\%$, $slot\_size = 2$ (d) $\alpha = 10\%$, $slot\_size = 5$

Fig. 4 Comparison of accuracy for TSP, pHeavy, and APPR at different α% and $slot\_size$.

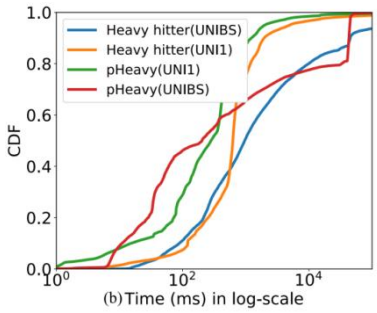


Fig. 5: Comparison of prediction times for Heavy Hitter Detection and pHeavy.

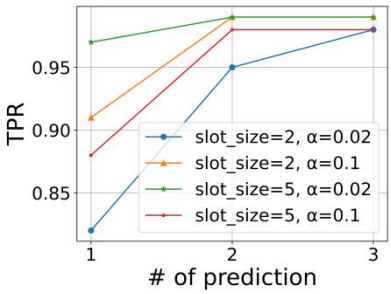


Fig. 6: Relationship between number of predictions and TPR for TSP.

predominantly consisting of TCP traffic and UNI2 consisting of UDP traffic.

Parameter Setting. TSP defines heavy hitter detection with three variables: heavy hitter occupancy (α), time slot size (*slot_size*), and the index of slot (*slot_index*). The *slot_size* determines the duration for heavy hitter prediction. Larger *slot_size* allows for more comprehensive flow information collection but incurs higher resource overhead. We conducted experiments with *slot_size* values of 2 seconds and 5 seconds. The heavy hitter threshold (α) affects both the number of detected heavy hitters and prediction accuracy. We tested with α values of 2% and 10% to cater to different network environments. TSP predicts combinations of *slot_size* and α from 1 to 4 *slot_index*.

B. Comparison of Accuracy

We evaluate the prediction accuracy of TSP against pHeavy and APPR using three metrics: TPR, TNR, and F1-Score, on the UNI1 dataset. The experiments are conducted at two occupancy rates (α = 2% and α = 10%), with TSP evaluated using two *slot_size* values (2s and 5s) and four *slot_index* values under the same conditions.

Figure 4 shows that all three algorithms exhibit high TPR and TNR, indicating effective detection of both heavy hitters and non-heavy hitters. However, the F1-Score varies across the methods and conditions.

At α = 2% (Figures 4a and 4b), TSP achieves a superior F1-Score compared to pHeavy and APPR. This is due to TSP's ability to leverage a higher number of heavy hitters per slot, which enhances model training and prediction performance by providing more balanced data.

At α = 10% (Figures 4c and 4d), pHeavy and APPR demonstrate higher F1 scores. This improvement is attributed to pHeavy's flow memory management, which filters out flows meeting certain termination conditions, and APPR's extensive feature set derived from application layer protocol interactions. However, APPR's complexity makes it less suitable for real-time data plane implementation. The lower F1-Score for TSP is primarily due to data imbalance caused by fewer heavy hitters per slot, which hampers the training process and leads to incorrect classification of negative samples. Although TSP maintains a high TPR, its TNR is lower than that of pHeavy and APPR, highlighting the challenge of correctly identifying non-heavy hitters under higher occupancy rates.

The decline in F1-Score for pHeavy and APPR as α decreases from 10% to 2% is notable. In contrast, TSP's F1-Score increases due to the higher number

of heavy hitters per slot, which helps mitigate data imbalance and leads to more accurate predictions.

C. Comparison of Prediction Time

We compare the prediction times of TSP, pHeavy, and Heavy Hitter Detection within programmable switches. Heavy Hitter Detection maintains a counter for each flow and sets thresholds equivalent to the position of the last decision tree in pHeavy (i.e., the 20th decision tree).

TSP predicts heavy hitters based on time slots, which constrains changes within specific slots. Consequently, a direct comparison of prediction times between TSP and pHeavy is not feasible. However, TSP's *slot_size* indirectly represents prediction time, allowing for a comparison based on the number of slots required for prediction.

Heavy Hitter and pHeavy derive results through initial packet analysis, making them suitable for temporal comparisons. The methods Heavy Hitter and pHeavy derive results through initial packet analysis, making them suitable for temporal comparisons. The heavy hitter prediction time is defined as the interval from receiving the first packet of a flow to predicting it as a heavy hitter. As shown in Figure 5, pHeavy predicts 90% of flows with an average prediction time of 2.6 seconds, while Heavy Hitter averages 7.9 seconds. TSP predicts at least 80% of large network flows on the first slot.

TSP's prediction time is inherently tied to its *slot_size* configuration. This allows TSP to confine predictions within set time frames, but it also means that the prediction time is quantized in units of *slot_size*. Consequently, the number of predictions correlates with prediction time. Figure 6 illustrates the relationship between the number of predictions and the TPR for TSP. The results show that 80% of heavy hitters are accurately predicted in the first prediction cycle, achieving up to 98% accuracy after three prediction cycles.

The significant differences in prediction times across Heavy Hitter Detection, pHeavy, and TSP can be attributed to their underlying mechanisms. pHeavy leverages initial packet inspection for rapid prediction, resulting in quick detection times but potentially less accuracy in highly dynamic network environments. Heavy Hitter Detection, while thorough, incurs longer prediction times due to its comprehensive evaluation, enhancing accuracy but may be impractical for real-time applications where swift decision-making is crucial.

TSP's slot-based approach ensures regular updates, making it suitable for environments where timely but not instantaneous predictions are required. The iterative refinement in TSP's predictions, shown by the increase in TPR from 80% in the first prediction to 98% by the third prediction, highlights its strength in leveraging accumulated data over successive intervals.

TABLE III
Hardware resources consumption of TSP on *Intel Tofino*.

Resource	Total / Average
ALU	13 Unit (29.55%)
Hash Bit	584 Bit (12.76%)
Hash Dist Unit	11 Unit (16.67%)
SRAM	160 KB (1.25%)
TCAM	3 Mtrits (58.33%)
Logical TableID	10 Unit (6.25%)
PHV	1156 Bit (18.82%)

D. P4 Hardware Switch Experiment

We have implemented TSP on P4 programmable switches equipped with the *Intel Tofino* ASIC. Both *Intel Tofino* switches and the *BMv2* pipeline support the P4 language, yielding comparable experimental results. However, unlike software simulations such as *BMv2*, deploying TSP on *Intel Tofino* necessitates careful consideration of hardware resource consumption, as shown in Table III. To schedule programs with dependency chains within the limited pipeline stages efficiently, TSP employs MAUs to consolidate chained operations and implement multi-branching decision trees. This strategy results in high TCAM resource usage (Average 58.33% per stage), whereas SRAM resources are relatively conserved (Average 1.25% per stage), as only the most recent stream information is stored.

V. CONCLUSIONS

In this paper, we introduced TSP, a system designed to detect heavy hitters in network traffic by leveraging both the Control Plane and the Data Plane. The Control Plane utilizes machine learning techniques to train a predictive model, addressing

data imbalance and optimizing for accuracy. This model is then translated into P4 code and deployed on *Mininet* with *BMv2* switches and *Intel Tofino* switches for real-time traffic analysis. Our evaluation, using real-world datasets, demonstrates that TSP achieves high prediction accuracy and efficiency, outperforming existing methods such as pHeavy and APPR in certain scenarios. The unique time slot-based approach of TSP enables it to maintain a balance between prediction accuracy and latency, making it adaptable to various dynamic network environments. Future work will explore further optimization of feature extraction and model training processes to enhance the robustness and scalability of TSP.

REFERENCES

[1] Liang Yang, Bryan Ng, and Winston KG Seah. Heavy hitter detection and identification in software defined networking. In International Conference on Computer Communication and Networks (ICCCN), pages 1 – 10. IEEE, 2016.

[2] Jing Ren, Yongwen Wang, Yiyun Liu, Xiong Wang, and Sheng Wang. Server-assisted traffic measurement for programmable data center networks. IEEE Transactions on Network Science and Engineering, 2024.

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

[4] Thiago Henrique Silva Rodrigues and F ´ abio Luciano Verdi. Detectingheavy hitters in network-wide programmable multi-pipe devices. In IEEE Network Operations and Management Symposium, pages 1 – 5. IEEE, 2024.

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

[6] Xiaochen Li, Weiran Liu, Jian Lou, Yuan Hong, Lei Zhang, Zhan Qin, and Kui Ren. Local differentially private heavy hitter detection in data streams with bounded memory. Proceedings of the ACM on Management of Data, 2(1):1 – 27, 2024.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review, 44(3):87 – 95, 2014.

[8] Frederik Hauser, Marco H ¨ aberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with p4: Fundamentals. Advances, and Applied Research. arXiv, 2101, 2021.

[9] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In USENIX Symposium on Networked Systems Design and Implementation (NSDI), volume 10, pages 89 – 92. San Jose, USA, 2010.

[10] 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 Conference, pages 339 – 350, 2010.

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

[12] 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.

[13] Xiaoquan Zhang, Lin Cui, Fung Po Tso, and Weijia Jia. pheavy: Predicting heavy flows in the programmable data plane. IEEE Transactions on Network and Service Management, 18(4):4353 – 4364, 2021.

[14] Coralie Busse-Grawitz, Roland Meier, Alexander Dietm ¨ uller, Tobias B ¨ uhler, and Laurent Vanbever. pforest: In-network inference with random forests. arXiv preprint arXiv:1909.05680, 2019.

[15] 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.

[16] 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 Conference, pages 254 – 265, 2011.

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

[18] 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.

[19] Ivan Tomek. A generalization of the k-nn rule. IEEE Transactions on Systems, Man, and Cybernetics, (2):121 – 126, 1976.

[20] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In Proceedings of the ACM workshop on hot topics in networks, pages 25 – 33, 2019.

[21] Wei-Yin Loh. Classification and regression trees. Wiley interdisciplinary reviews: data mining and knowledge discovery, 1(1):14 – 23, 2011.

[22] mininet. [Online]. Available: https://github.com/mininet/mininet, 2021.

[23] p4lang. Behavioral model (bmv2). [Online]. Available: https://github.com/p4lang/ behavioral-model, 2022.

[24] Intel® tofino™ series programmable ethernet switch asic. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[25] UNI datasets. [Online]. Available: http://netweb.ing.unibs.it/ ~ ntw/tools/traces/.