Dryad: Deploying Adaptive Trees on Programmable Switches for Networking Classification

Guorui Xie[1,2], Qing Li[2], Jiaye Lin[1,2], Gianni Antichi[3,4], Dan Zhao[2], Zhenhui Yuan[5], Ruoyu Li[1,2], Yong Jiang[1,2]
[1] International Graduate School, Tsinghua University, Shenzhen, China [2] Peng Cheng Laboratory, Shenzhen, China
[3] Politecnico di Milano, Italy [4] Queen Mary University of London, UK [5] Northumbria University, UK

2023 IEEE 31st International Conference on Network Protocols (ICNP) | 979-8-3503-0322-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICNP59255.2023.10355629

***Abstract*—Decision trees (DT) have been used for high-speed networking classification on programmable switches. Most DT solutions, however, are static and cannot be deployed once the switch resource changes. In this paper, we propose Dryad to fast reprogram tree models when resource budgets change. In Dryad, we first develop a large and accurate "one-training-for-all" DT (ODT) that can be quickly resized without computational retraining. ODTs are deployed in switches using a progressive search algorithm that searches the adaptations according to their resources. To achieve high accuracy and low packet latency, the adaptation leverages 1) innovative hard and soft pruning methods to compress the ODT rapidly with minimal performance loss; and 2) P4 scaling operations of match-action table arrangement and joint range-ternary match, which allow the switch to accommodate a larger (i.e., more accurate) ODT. Finally, an ODTCompiler is proposed to automatically convert the adapted ODT into a P4 program and then install it. Experimental results on three commodity switches under different resource scenarios show that Dryad achieves a higher classification F1-score (3.78% higher), and completes the adaptation $161\times$ faster than other solutions.**

I. Introduction

Over the past few years, machine learning (ML) models have been applied to many networking classification tasks, e.g., malware detection [1]–[3], traffic classification [4]–[7], and flow size prediction [8]. Usually, the conventional solutions deploy models on high-performance X86 servers, which cannot provide satisfactory processing latency and capacity [9]. Modern programmable switches (e.g., P4 switches [10], [11]) support Tbps-based custom packet processing, which provides new alternatives for ML models, i.e., in-network intelligence [12]. Programmable switches are typically equipped with two types of chips (CPUs and ASICs). The CPU can run control APIs to configure network applications, e.g., compiling and installing P4 programs to the ASIC. The ASIC (e.g., Tofino [13] and FlexCore [14]) is of Tbps and follows a match-action paradigm. Match-action tables form the P4 program and are spread on the ASIC pipeline. Each table maintains a set of key-action entries. When a packet (e.g., its header) is matched by a table entry, the associated action can conduct simple operations like assignment or integer addition/subtraction.

The decision tree (DT) model naturally coheres with such an ASIC's match-action paradigm due to its inherent rule-matching nature in the classification [15]–[17]. As such, the DT has now become prevalent in in-network intelligence solutions [9], [18]–[21]. Even though existing works have made significant progress, they do not focus on one important requirement: rapid adaptation to changes in switch resources. In our view, switches distributed in the network can have differing resource requirements. For example, switches can run applications such as NAT and routing at the same time. The corresponding switches should allocate different resources for such applications, resulting in changes in the resource budget for the DT deployment. Even if the budget for a switch is set, it may still change later. Consider the DT deployed for measurement applications like ESketch [22], ESketch recommends that the memory allocated be increased dynamically with the number of observed flows. This also affects the available resources for the DT.

A naive approach to adapt a DT to the available resources is to maintain different DT models for different resource constraints at a central controller. But such an approach can be inefficient and slow. Retraining different DT models on the massive training dataset is inefficient in both computation and storage. Furthermore, the communication latency between the switch and the controller can be non-negligible when more switches request different adaptations. To quickly adapt the running P4 programs to the resources with less overhead, one can use control APIs on switch CPUs, e.g., the fast refresh in Tofino ($< 50ms$) [23], or new runtime architectures like FlexCore [14] and IPSA [24]. However, this is not sufficiently fast for in-network intelligence which has to first adjust the DT and then reprogram it as a P4 program.

In this paper, our key insight is that instead of maintaining different DT models for different resource constraints, we can augment a trained DT with the statistical distribution of various classes at each node. We refer to this augmented DT as a *"one-training-for-all" DT (ODT)*. To adapt to different resource constraints, we prune this augmented DT to derive a DT that can achieve good performance under a given set of resource constraints. We implement our approach in a system called *Dryad*[1]. Dryad is *fast*, because we only need to train the ODT on our dataset once. Thereafter, we derive the required DT model by pruning the ODT without the need to process the dataset again. Dryad is *accurate*, as we not only adjust the model size for resource constraints, we also implement scaling

Corresponding author: Qing Li (liq@pcl.ac.cn)

979-8-3503-0322-3/23/$31.00 ©2023 IEEE

[1]Dryad is the tree nymph (spirit) in Greek mythology.

operations in P4 to allow us to fit deeper DT models to increase accuracy at the cost of slightly increased latency. Moreover, the Dryad pipeline is fully *automated*. The entire process of model tuning, P4 program generation, and installation is automatically completed inside the switch CPUs to boost the adaptation efficiency. In summary, we make the following contributions:

- We propose the *"one-training-for-all" DT (ODT)*, where we augment each node of a DT to summarize the observed statistics of the dataset during the training (i.e., tree growing). Based on these statistics, the size of the ODT can be efficiently pruned on switch CPUs, without any additional computation or retraining on the dataset.
- We propose a *progressive search algorithm*. Subject to available resource constraints, the algorithm traverses all feasible combinations of the pruning (to reduce the demand) and P4 scaling operations (to squeeze the supply) to find the adaptation that obtains the max reward on both classification accuracy and processing latency.
- We implement an *ODTCompiler* to automate the deployment. According to the optimal ODT adaptation, the ODTCompiler generates the desired P4 code (i.e., match-action tables along with table entries) and finally calls the built-in control APIs to directly install the P4 code on the switch ASIC pipeline.

We implement Dryad[2] and test it on three commodity switches [25]–[27]. Our experimental results reveal that: 1) Our pruning-based resizing is cost-effective. Compared with retraining using the full data set, the pruning-based ODT adaptation is $161\times$ faster; 2) The progressive search algorithm scales well and can find the optimal adaptations for different switches. We demonstrate that the adaption process can be finished on switch CPUs within $1.61s$ in Python (which can likely to reduced to $\sim 55ms$ in C++ [28]).

II. Background

A. Traditional Learning-based Networking Classification

Machine learning (ML) algorithms have long been applied to various classification problems in networks. In [1], the authors present KitNET, a malicious traffic detection solution that uses an ensemble of neural networks to collectively differentiate the benign and malicious traffic. The work in [2], [3] exploits learning algorithms (e.g., support vector machine, fully-connected neural network) for malicious traffic detection. In [4], the authors design a learning-based system for the traffic characterization and the application identification tasks using deep learning models, i.e., convolution neural network (CNN) and stacked auto-encoders (SAE). In [5], the authors present the byte segment neural network (BSNN), which is based on the bidirectional recurrent neural network (RNN), for network traffic classification. Also targeting the problem of traffic classification, [6], [7] leverage deep learning algorithms such as one-dimensional CNN and self-attention mechanism. In [8], the authors address the flow size prediction problem by employing learning models such as Gaussian process regression (GPR) and neural network (NN).

Although these ML-based networking applications have outperformed traditional solutions, they are generally trained and deployed on powerful X86 servers. As a result, the time and capacity costs associated with collecting traffic and analyzing it on these devices are unacceptable [9], [12].

B. Characteristics of the P4 Switch

Unlike X86 servers, programmable switches, e.g., the P4 switches, support customized packet processing logic with low latency [10], [11], showing the promise of empowering these ML-based solutions on the data plane. Generally, each P4 switch contains two types of processors. One is an ordinary CPU, which acts as a local controller and can be responsible for the management of the switch's data plane. The network administrator can install P4 programs and table entries through control APIs on the CPU. In addition, switch architectures such as FlexCore [14], IPSA [24], and Tofino [13] also provide new APIs to fast reconfigure running P4 programs in milliseconds or even runtime, with minimal traffic disruption.

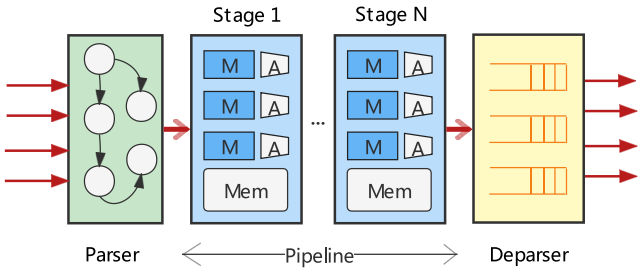


Fig. 1: The P4 switch data plane.

The other is a high-speed ASIC (e.g., Tofino), acting as the data plane of Tbps [13]. Fig. 1 illustrates the key components of an example Tofino data plane. Incoming packets are first mapped into Packet Header Vectors (PHV) by the parser. Then, the PHVs are passed to the pipeline consisting of several stages. At each stage, headers in the PHV may match (M) a given table entry stored in the memory (Mem) block, triggering the associated action (A), i.e., the match-action paradigm. Note that stages are the basic resource units. Memory is uniformly distributed amongst different stages, and no stage can access a memory block in another. Finally, the modified PHVs are reorganized into packets by the deparser. Two important types of matches for the packets are: 1) range match. The key value should fall between a lower limit and an upper limit specified by a rule; 2) ternary match. Each rule is associated with a pair of (*Mask, Value*). The key value is first *AND*ed with the *Mask*, and then compared with the *Value* for equality. Nevertheless, to guarantee high-speed processing, instructions in actions are strictly limited to integer operations like addition/subtraction or assignment, which hinders the switches' programmability with sophisticated learning models like neural networks [29], [30].

[2]The code is available at https://github.com/xgr19/Dryad.

C. DT Techniques and In-network Intelligence

Unlike other complex models, tree models are rule-based classifiers and naturally suit the match-action in P4 switches. There are several DT variants, e.g., ID3 [15], C4.5 [16], and CART [17]. One main difference among these variants is the node/dataset splitting criterion in the training. For example, C4.5 uses the normalized information gain while CART proposes the Gini index. For the sake of convenience, several in-network approaches [9], [18]–[21] are proposed to completely deploy the CART model on switch ASICs to achieve line-speed classification, as CART has been well-implemented in the famous scikit-learn library [31], [32].

The authors in [9] design IIsy to convert a DT into a P4 program via a *feature-decision* manner. For a DT with k input features, IIsy utilizes $k+1$ tables for the conversion. Each of the first k feature tables checks the value of a specific feature via the match paradigm, and then encodes the match result into a metadata field in the PHV by the action paradigm. For the last table (aka decision table), the metadata fields are used as keys to finally output the classification results. In [19], the authors propose SwitchTree to deploy the DT in a *level-table* manner. In SwitchTree, each pipeline stage allocates a table to embed a particular level of the DT. Similar to IIsy, SwitchTree also uses metadata fields in the PHV to act as both the match results and match keys in subsequent level tables. Some other works that also attempt to deploy trees on the switch ASICs are [20], [21]. These DT-based approaches have made significant progress, but one area that needs critical discussion is the quick reconfiguration of the DT based on the changing switch resources. Due to different network management tasks, switches may run different applications together with the DT (e.g., NAT, routing, and measurement), which leads to varying resources on pipeline stages and rule capacity. Although the recently extended version [18] of IIsy also considers the resource constraints in the design, the extended IIsy is still a static solution to deploy a fixed-size model on all devices, which may not scale well on switches of varying resources.

One may reconfigure the DT by retraining numerous DTs for different resource requirements on a central controller. However, retraining various models on massive traffic data (e.g., the UNSW-NB15 dataset [33] consists of 100GB traffic) is time-consuming and computationally expensive [34]. Also, it would increment the switch-controller communication overhead and delay the response when multiple switches are concurrently requesting different adaptations. To make network applications quickly reconfigurable for varying resources, the switch CPUs have already supported control APIs [14], [23], [24] to reprogram the P4 program. Hence, it may be possible to also offload the model resizing (especially the pruning-based DT resizing) to individual switches for efficiency gains. The idea of DT pruning is not new, e.g., [17], [35]–[37]. However, most of these solutions prune trees on the dataset. The post-pruning in [35] employs test cases in the dataset to estimate if the pruning of a node will reduce the accuracy. The authors in [36] use all samples in the training dataset to tailor the DT. Also, they are heuristics algorithms and cannot control the tree size (e.g., depth) precisely. In our view, a suitable pruning should be independent of the dataset to reduce switch overheads. Besides, we would like to be able to prune the tree to an arbitrary size to better meet different resource constraints.

III. Augmenting the DT for Efficient Pruning

As mentioned in §II-C, there are many DT variants available. For example, the CART [17] in scikit-learn [31], [32] is often used in in-network solutions [9], [18], [21]. While there are existing pruning techniques to limit DTs to a given maximum depth, we want to do more than limit the depth of a deployed DT to the number of stages in a P4 switch. By employing recirculation, we can effectively support DTs at a depth deeper than the number of pipeline stages available.

To support a pruning strategy that can take advantage of this feature, we augment the CART by adding statistical distributions of the various classes to each node during training. We refer to this augmented DT as a *"one-training-for-all" DT (ODT)*. Once an ODT is trained, it can be pruned efficiently on all switches instead of repeating the training process with the dataset, thus significantly saving the computation and storage on switches. Also, the performance of the pruned ODT can be easily estimated from statistics in the pruned DT (see §IV-B).

A. ODT Training

The training (or generation) of the ODT is similar to a regular CART [17], as shown in Algorithm 1. The training dataset D consists of m samples, each in the form of (x_i, y_i). $x_i \in \mathbb{R}^k$ represents sample i's feature values on the feature set F, and y_i is the class label of sample i. The goal of the training is to produce a DT that will partition the dataset into a set of leaves that have a unique class label. What we do differently from the regular DT training is that for each node n, we augment it by adding an attribute $values$. $n.values$ records the number of samples from each class for the samples seen at node n (see line 2).

The training algorithm is a recursive function applied to the training samples at each node as follows:

- If all samples in the node have the identical class label C, then this node is considered as a leaf with class label C, see lines 3∼5.
- A node is also identified as a leaf if all samples have the same value on every feature, even if not all the samples have the same class label. In this case, the class label of this leaf is set as the class with the majority of samples (implemented with the function MajorVote(.), see lines 6∼8). Note that Dryad does not limit the depth d of the tree during the training process to achieve maximal accuracy, without considering constraints on the model size.
- Otherwise, this means that not all the samples have the same value on every feature and we need to determine the optimal feature f^* and its threshold f_t^* with which to partition the samples into nodes at the next level.

Algorithm 1: ODT Training

Input: Train set $D = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)\}$; Feature set $F = \{f_1, \ldots, f_k\}$; Tree limited depth d.

Function: ODTGENERATE$(D, F, i = 0)$

1 Generate tree node n;
2 Summarize the number of samples a_i of each class i in D: $n.values = [a_1, \ldots, a_j]$;
3 **if** samples in D ALL belong to the same class C **then**
4 $\quad n.class \leftarrow C$; **return**
5 **end**
6 **if** samples ALL have the same value on each feature of F **OR** depth $i = d$ **then**
7 $\quad n.class \leftarrow$ MajorVote$(n.values)$; **return**
8 **end**
9 Find the optimal splitting feature f^* and its threshold f_t^*: $n.thresh \leftarrow f_t^*$;
10 Create two child nodes of n: $left\_child$ and $right\_child$;
11 **for** $n' \in \{left\_child, right\_child\}$ **do**
12 $\quad$ **if** $n' = left\_child$ **then**
13 $\quad\quad D_{n'} \leftarrow \{(x, y) \mid x.f^* \leq f_t^*, x \in D\}$;
14 $\quad$ **else**
15 $\quad\quad D_{n'} \leftarrow \{(x, y) \mid x.f^* > f_t^*, x \in D\}$;
16 $\quad$ **end**
17 $\quad$ **if** $D_{n'} = \varnothing$ **then**
18 $\quad\quad n'.class \leftarrow$ MajorVote$(n.values)$; **return**
19 $\quad$ **else**
20 $\quad\quad n' \leftarrow$ ODTGENERATE$(D_{n'}, F, i + 1)$;
21 $\quad$ **end**
22 **end**

Output: A trained ODT T_0 where internal nodes obtain the $thresh$ attribute and the leaves maintain the $class$ attribute.

To determine the optimal feature, we iterate through all the feasible feature-threshold combinations. In particular, because we are using the DT variant of CART, this refers to the feature and threshold combination that results in the lowest Gini index [17]. Once the optimal feature and thesholds are determined, the samples are partitioned into a left branch ($\leq f_t^*$) and the right branch ($> f_t^*$) (lines 9∼22).

After the training with Algorithm 1, the ODT can predict class labels for test samples. For a test sample, its feature value is compared with internal nodes' threshold f_t^* and then this sample is routed to the left/right child nodes until it reaches a leaf node, then the $class$ of the leaf is returned.

B. ODT Pruning

The ODT obtained from the training process described in §III-A would generally be too big to fit in a P4 switch. To reduce the depth of the ODT to fit within the resource constraints, we implement two pruning techniques: (i) *hard pruning* will forcefully reduce the depth of the ODT; and (ii) *soft pruning* which removes redundant nodes arising from hard pruning to make the ODT more compact.

Algorithm 2: Hard Pruning

Input: A tree with root r; A queue $S = [r]$ for nodes; A queue $V = [0]$ for nodes' levels; Tree limited depth d.

1 **while** $S \neq \varnothing$ **do**
2 $\quad n \leftarrow S$.pop() ; // queue is first-in-first-out
3 $\quad l \leftarrow V$.pop();
4 $\quad$ **if** n is an internal node **then**
5 $\quad\quad$ **if** $l < d$ **then**
6 $\quad\quad\quad S$.push$(n.left_child)$;
7 $\quad\quad\quad V$.push$(l + 1)$;
8 $\quad\quad\quad S$.push$(n.right_child)$;
9 $\quad\quad\quad V$.push$(l + 1)$;
10 $\quad\quad$ **else if** $l = d$ **then**
/* set n as a leaf, remove its subtree */
11 $\quad\quad\quad n.class \leftarrow$ MajorVote$(n.values)$;
12 $\quad\quad$ **end**
13 $\quad$ **end**
14 **end**

Output: Hard pruned ODT T_h.

Hard Pruning. The hard pruning sets internal nodes with a specific depth d as leaves, removing all nodes deeper down the tree. For each new leaf n, we decide its class label by conducting the majority vote on its $n.values$ attribute. Details of the hard pruning algorithm are shown in Algorithm 2. The core idea is that we first perform a breadth-first traversal on the tree, reach the desired depth d, and then set each internal node at this depth level to be a leaf by executing the MajorVote(.) function on its $values$ in line 11.

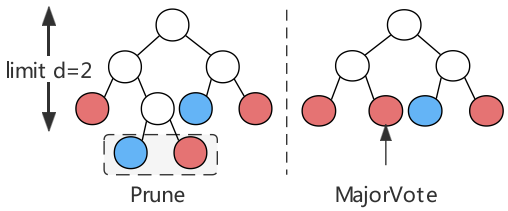


Fig. 2: An example of the hard pruning.

Fig. 2 shows an example of the hard pruning algorithm. The different colors (red and $blue$) of leaf nodes indicate different class labels. The two bottom leaves that exceed the depth limit $d = 2$ are both removed. We set their parent as a new leaf with the class label obtained through MajorVote($parent.values$). Notably, once the ODT is trained, each node is assigned an array $values$ (Algorithm 1 line 2). This array records the number of samples split to this node. Assume that $parent.values = [red : 10, blue : 5]$, then MajorVote($parent.values$) will return the class label of red. Nevertheless, in this example, the MajorVote(.) returns a leaf

that has the same class label as its siblings, resulting in redundant nodes (the following soft pruning will tackle this problem).

Algorithm 3: Soft Pruning

Input: A tree with root r; A list $L = []$ for leaves' class labels; A stack $S = [r]$ for nodes.

```
1  while S ≠ ∅ do
2      while S.top.left_child ≠ NULL AND ≠ visited do
3          S.push(S.top.left_child);
4      end
5      n ← S.pop() ;   // stack is last-in-first-out
6      if n is an internal node with l leaves then
7          if L[−l : −1] are ALL the same class C then
8              n.class ← C ;        // set n as a leaf
9              L.add(n.class);
10         end
11     else if n is a leaf then
12         L.add(n.class);
13     end
14     if S.top.right_child ≠ NULL AND ≠ visited then
15         S.push(S.top.right_child);
16     end
17 end
```

Output: Soft pruned ODT T_s.

Soft Pruning. In Dryad, we propose a soft pruning algorithm to remove redundant nodes on the ODT. As shown in Algorithm 3, we use the post-order traversal to recursively visit the tree. In this algorithm, the left subtree of the current node is traversed first (lines 2∼4), followed by the right subtree (lines 14∼16), and then the current node is visited. If all leaves of the current node have the same class label C, we remove all branches of the current node and set the current node as a leaf with label C (lines 7∼10).

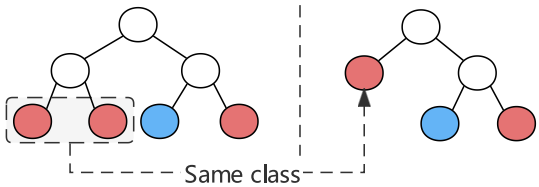


Fig. 3: An example of the soft pruning.

Fig. 3 illustrates an example of the soft pruning algorithm. As can be seen, the leftest white internal node has two leaves that have the same red class. Soft pruning removes these redundant leaves, and makes their parent internal node a leaf by simply assigning it the red class label. It should be clear that the soft pruning does not affect the ODT's classification performance.

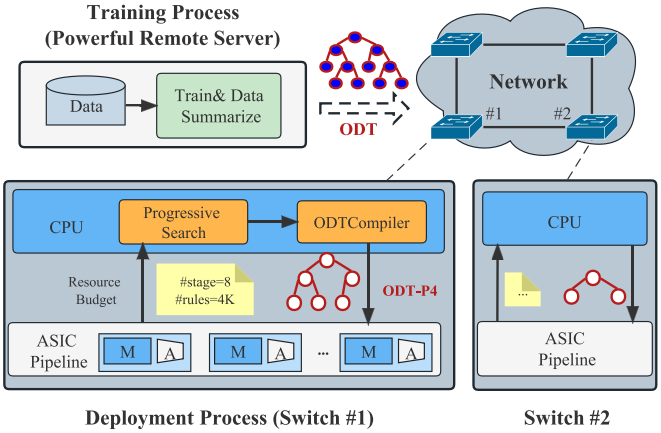


Fig. 4: The Dryad framework.

IV. Dryad Implementation

An overview of the Dryad system is shown in Fig. 4. Dryad consists of two independent stages: a training process that computes the ODT from the dataset and an *adaptive* deployment process that determines how the ODT should be pruned and deployed on the P4 switch.

Training process. Training, as described in §III, is performed on a high-performance remote server, which can handle computationally expensive training and massive datasets. Upon receiving a dataset, we endeavor to train an accurate ODT without compromising model size due to resource constraints. Especially, nodes in the ODT have summarized the observed training data statistics for future adaptations.

Deployment process. The deployment process operates independently at programmable switches with the assistance of built-in control APIs. Whenever the resource constraints change, Dryad might adjust the ODT at the potential cost of lower accuracy in two possible ways:

1) Reduce the demand. In particular, we compress the ODT using hard pruning, which prunes the ODT to a specific depth, and soft pruning, which further removes redundant nodes without sacrificing classification accuracy (§III-B). Our proposed pruning mainly relies on the summarized dataset statistics in the ODT, which is fast and efficient.
2) Increase the supply. We propose two P4 scaling operations to maintain a larger ODT of higher accuracy (§IV-A). As part of the table implementation, we can support multiple ODT level tables with a single pipeline stage by allowing packets to loop through the pipeline in multiple rounds, so that more levels can be hosted in the pipeline. For table entries' implementation, we jointly use the range and ternary match to reduce the capacity required by each classification rule in the ODT.

These two perspectives result in a trade-off between accuracy and packet process (classification) latency. To achieve higher accuracy, a larger ODT is preferred; however, to support a larger ODT, we will require more P4 scaling operations that will result in higher processing latency.

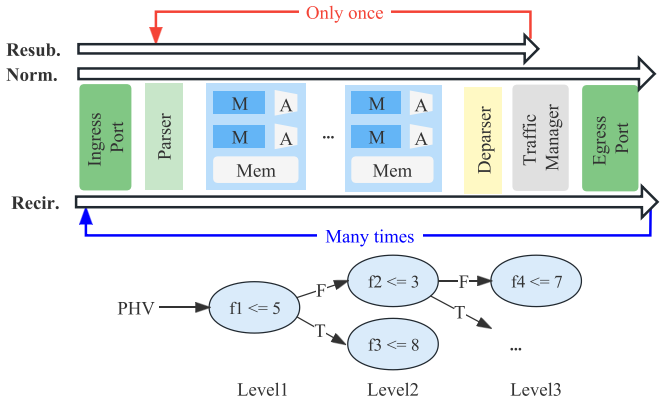


Fig. 5: Three ODT implementation options in the ASIC[3]. In the ODT branches, "T" is True, and "F" is False.

We run a progressive search algorithm (§IV-B) to determine the optimal configuration for the abovementioned pruning methods and P4 scaling operations. During the search, we try to maximize an objective function that trades off classification performance against packet process latency. The classification performance is estimated from the summarized statistics, without needing any processing of the training dataset. Furthermore, we describe optimization techniques in §IV-B that can speed up the implementation of this algorithm on switch CPUs.

After obtaining the optimal configuration, the ODTCompiler (§IV-C) automatically adjusts the ODT and converts it into a P4 program of match-action tables (along with table entries), and finally calls the built-in control APIs to quickly reinstall the application in the switch ASIC pipeline.

A. P4 Scaling Operations

1) Match-Action Table Arrangement: There are several ways to map a tree model on a P4 switch's ASIC [9], [19]–[21]. Currently, we select the level-table manner [19]–[21] (discussed in §II-C), which makes it easier to associate the tree depth with the pipeline stages, as the base of our ODT deployment. In a switch ASIC like Tofino (Fig. 5), a packet traverses multiple consecutive pipeline stages (the light blue blocks) in the form of PHV (i.e., packet header vector in §II-B). Each stage can read and modify the packet properties and affect the processing of the next stage through its stored match-action tables. For the ODT, each stage can maintain a level of the ODT as a match-action table, and the PHV conveys the input feature values. We call this approach the **normal solution** (Norm.). As an illustration, $Level1$ (one node) is placed in the first stage, and $Level2$ (two nodes) is in the second stage. This solution enjoys low packet process latency because the classification results can be obtained directly when packets pass through the pipeline. However, this solution is only feasible when the number of the ODT's levels does not exceed the number of available stages.

To support a deeper ODT, we propose the resubmit and recirculate implementation options, which host two or more ODT levels in a stage by allowing a packet to go through the pipeline more than once. As shown in Fig. 5, with the help of the built-in traffic manager (TM), the **resubmit solution** (Resub.) loops the packet back to the parser, and then forces the packet to pass through the pipeline again. Note that the TM automatically does the resubmit operation once the $resubmit\_type$ flag in the PHV is non-zero. That is, at the last pipeline stage, all we do is set $resubmit\_type$ and embed some necessary metadata (e.g., the last visited level, the corresponding tree node, and the match result) into the PHV. In the **recirculate solution** (Recir.), we first defined ingress port #68 and egress port #68 to be looped. Then, by assigning the output port as 68 at the last stage, and embedding the necessary metadata in the PHV, the packet can be redirected to the ingress multiple times. While resubmit and recirculate improve resource utilization and accommodate more ODT levels, they also add to the packet process latency as a packet must be sent through the pipeline two or more times.

2) Joint Range-Ternary Match: To predict an unclassified sample, the trained ODT is traversed through a series of threshold comparisons. Starting from the root, at each internal node, the value of the sample's splitting feature f^* is compared with the threshold f_t^*. The comparison outcome leads the sample to either the left ($f^* > f_t^*$) or the right branch ($0 \leq f^* \leq f_t^*$). After hitting a leaf node, a class label is assigned to the sample.

As discussed in §II-B, there are two important match types (range and ternary) defined in P4. It is most straightforward to use the range match to represent these range comparisons. However, given a fixed memory block in a stage, we observe that using the range match will lead to much fewer allowed table entries than the ternary match. But simply using the ternary match is not efficient because each range (e.g., $0 \leq f^* \leq f_t^*$) must be broken into several $(Mask, Value)$ pairs [38]. Thus, in our match-action tables (ODT levels), we use the range match and ternary match jointly. That is, the range match for features whose number of bits exceeds a threshold bit, while the ternary match for features whose bits are below the bit.

B. Progressive Search Algorithm

The key idea of our progressive search algorithm is to determine the optimal configuration for deploying an ODT in terms of the following 3 parameters: (i) the optimal number of rounds that a packet needs to go through the pipeline, which determines the solution (among Norm., Resub., and Recir. in Fig. 5) used for placement of the ODT level tables; (ii) the optimal bit threshold for choosing between the range match and the ternary match for the ODT feature comparison in level tables; and (iii) the optimal hard pruning depth. We iterate the search space for these three parameters to determine the

[3]Tofino ASICs restrict that resubmit can loop packets back only once through the traffic manager so as to reduce the packet latency. In contrast, recirculate does not have a limit on the number of loopbacks.

Algorithm 4: Progressive Search Algorithm

Input: ODT $tree_0$; Stage budget $stages$; Max packet process rounds k; A coefficient α to balance the ODT accuracy and packet process latency.

1 Initialize a reward list S with k zeros;
2 Initialize a strategy list $strategy$ with k NULL;
3 **for** $round$ from 1 to k **do**
4 $tree \leftarrow \text{deepcopy}(tree_0)$;
5 **for** $estdepth$ from $round \times stages$ to 1 **do**
6 $tree \leftarrow \text{Soft}(\text{Hard}(tree, estdepth))$;
7 **for** bit from 0 to the max bits of tree features **do**
 /* the rule capacity per stage if features with bits $\leq bit$ use ternary match */
8 $rulecap \leftarrow \text{RuleEst}(bit)$;
9 $real\_depth \leftarrow \text{Encode}(tree, rulecap, bit)$;
10 $reward \leftarrow \alpha\text{MetricEst}(tree, real\_depth) + (1-\alpha)\frac{1}{\text{LatencyEst}(round)}$;
11 **if** $reward > S[round]$ **then**
12 $S[round] \leftarrow reward$;
13 $strategy[round] \leftarrow (real\_depth, bit, round)$;
14 **end**
15 **end**
16 **end**
17 **end**

Output: The strategy $(real\_depth, bit, round)$, whose corresponding $S[round]$ is the maximum.

appropriate trade-off between accuracy and packet processing latency (Algorithm 4).

1) Algorithm Details: The following is a brief overview of the three nested for loops:

- The outermost loop iterates through the different rounds (i.e., $round$) that packets loop through the pipeline. In this loop (line 4), we always copy a new tree from the original full-depth $tree_0$ per $round$ so as to reuse $tree_0$ during the search.
- The middle loop searches through all feasible values of hard pruning depth $estdepth$ that may be arranged in the pipeline by $round \times stages$. Soft(.) and Hard(.) refer to soft and hard pruning, respectively. We see in line 6 that the ODT is first hard pruned to the depth $estdepth$, then soft pruned to remove its redundant leaves.
- The innermost loop iterates through the value range of bit, which is the decision threshold to choose between the range match and the ternary match (see §IV-A2).

In the innermost loop, we first determine the number of allowed table entries at each stage with RuleEst(.) in line 8 for the specified bit threshold. RuleEst(.) is essentially a lookup table that we had determined empirically. Next, we determine the resulting depth of the ODT after encoding the tree into match-action tables using Encode(.) in line 9. *Reward* is the optimization objective function for each configuration that jointly considers the ODT performance and packet process latency.

The ODT performance (i.e., accuracy) is calculated as follows:

$$\text{MetricEst}(.) = \frac{\sum \max(l_i.values)}{\sum l_i.values} \times 100\%$$

where l_i are the leaves in the ODT and $\max(l_i.values)$ is the number of samples in the majority class for leaf l_i. From Algorithm 1, the class label for leaf l_i is the class with the most samples at the leaf. Hence, numerator $\sum \max(l_i.values)$ is the number of correctly classified samples, and the denominator is the total number of samples. For example, for an ODT of two leaves ($l_1.values = [class_0 : 10, class_1 : 3]; l_2.values = [class_0 : 1, class_1 : 5]$), the predicted accuracy is $\frac{10+5}{10+3+1+5} \times 100\%$. Note that MetricEst(.) can be modified to support other metrics such as precision, recall, and F1-score, as all of them can be obtained using $values$. We leave this as future work.

The packet process (classification) latency is obtained from LatencyEst(.), which is determined empirically from measurement. The values for LatencyEst(.) will depend on the hardware platform.

As shown in line 10, the accuracy and packet process latency are weighted by α and $1-\alpha$, respectively. By assigning different values (from 0.0 to 1.0) to α, we can make the appropriate trade-off between these two objectives. For example, when the traffic rate is slow (10Gbps), adding some packet latency to achieve high accuracy is acceptable. Thus, we can use a large α (see §V-A for details).

Upon finishing iterating through the above triple nested for loop, an optimal configuration $(real\_depth, bit, round)$ with the maximum reward is obtained. Here, $real\_depth$ indicates how deep an ODT is to be pruned; bit is the optimal threshold for the selection of range match and ternary match; and $round$ specifies which of the following three table arrangement solutions should be used: Norm. ($round = 1$), Resub. ($round = 2$), or Recir. (for $round \geq 3$).

2) Algorithm Optimizations: We propose two optimizations to accelerate Algorithm 4, so that it can run on ordinary switch CPUs. **The early-stopping**: For the outermost loop in Algorithm 4, the resource consumption increases linearly as $round$ increases. Therefore, while iterating through the outermost loop, whenever the candidate strategy exceeds the maximum resource budget, we stop examining larger $round$ to avoid taking up excessive resources. **The multi-core**: CPUs are now equipped with multi-core technology that allows the system to perform multiple tasks concurrently with higher overall system performance. For each $round$ in the outermost loop, we generate an independent process running on a CPU core alone. Moreover, an early-stopping signal is used in inter-process communication [39].

C. ODTCompiler

After Algorithm 4, the ODT is passed to the ODTCompiler. The ODTCompiler first uses both soft and hard pruning algorithms to prune the ODT to the depth of $real\_depth$ (thus $real\_depth+1$ levels). Then, ODTCompiler instantiates a P4 program from the appropriate pre-installed P4 code templates according to $round$. Next, the ODTCompiler traverses the nodes in each ODT level, encoding them into table entries with the range match or ternary match according to bit. At last, the P4 program (the instantiated code along with the table entries) is compiled and installed on the ASIC pipeline. We mainly discuss two key processes: P4 code instantiation and table entry encoding below.

1) P4 Code Instantiation: Our ODTCompiler maintains several P4 templates:

- The *base.p4* template defines a universal P4 program that consists of the standard components like the parser, the pipeline, and the deparser.
- The *norm.p4*, *resub.p4*, and *recir.p4* inherit the universal base template and then add their required modifications accordingly (see §IV-A1).

When the ODTCompiler is called, it instantiates a P4 program, namely, *p4app.p4*, by selecting the appropriate template. Then, according to the adaptation strategy $(real\_depth, bit, round)$ and the received ODT, the ODTCompiler decides how many match-action tables are derived in the template program, which PHV elements (i.e., ODT features) are used as table keys, what kind of matching (e.g., range or ternary) is used, etc.

List 1 shows the code fragment of a template switch pipeline in *base.p4*. This pipeline is defined by the `control` keyword along with multiple meta parameters (lines 1∼6). Except for the P4 code, we use the Jinja[4] syntax to write the special placeholders in this template which allows the ODTCompiler to replace with meaningful words. For example, the placeholders `resubmit1` (List 1 line 13) is overwritten by the *resub.p4* in List 2 lines 2∼8.

2) Table Entry Encoding: After finishing the P4 code generation, the ODTCompiler will encode the ODT into table entries. This encoding process can be done by a breadth-first traversal on the ODT. List 3 shows a part of it. For example, in line 3, we use two queues to store the tree nodes in the current level and the next level. For each node in the current level, we examine whether it is a leaf or not. If so, lines 9∼13 will encode it to a table `entry` with the action of class label assignment. In `entry`, `p4app` is the corresponding P4 program name, `level%d` indicates which level table this entry belongs to, and `SetClass` is the associated P4 action for this rule which assigns a class label (i.e., `pred_cls_id`) to a matched packet.

Listing 1: The code fragement in *base.p4*

```
1  control Ingress(
2      // User
3      inout my_ingress_headers_t
                                  hdr,
4      inout my_ingress_metadata_t
                                  meta,
5      // Intrinsic
6      inout ingress_intrinsic_metadata_for_tm_t
                 ig_tm_md)
7  {
8      // initialized level tables
9      {% for level in levels %}
10     Level(){{level}};
11     {% endfor %}
12
13     {% block resubmit1 %}{% endblock %}
14
15     apply {
16         {% for level in levels %}
17         {{level}}.apply(hdr,meta,ig_tm_md);
18         {% endfor %}
19     }
20 }
```

Listing 2: The code fragement in *resub.p4*

```
1  {% extends "base.p4" %}
2  {% block resubmit1 %}
3  action action_packet_add_info(){
4      meta.resubmit_data.node_id = meta.node_id;
5      meta.resubmit_data.compare_result = meta.
           compare_result;
6      ig_dprsr_md.resubmit_type = 2;  // previously
            discussed in §IV-A1
7  }
8  {% endblock %}
```

Listing 3: Python code for encoding table entries

```
1  def export_p4_rules(tree) {
2      # store node ID per level
3      queues = [queue(), queue()]
4      # omit other code...
5      for level in range(0, tree.depth):
6          cur_level = queues[level%2]
7          while len(cur_level) > 0:
8              node_id = cur_level.popleft()
9              if tree.feature[node_id] == None:
10                 # leaf has no f_i^* for comparision
11                 pred_cls_id = argmax(tree.values[
                       node_id])
12                 entry = 'bfrt.p4app.pipe.level%d.
                       '+
13                   'add_with_SetClass(%d, %d)'%(
                         level,node_id,pred_cls_id)
14             else:
15                 # omit to encode inner nodes...
16                 # add child nodes of next level
17                 queues[(level+1)%2].append(tree.
                       children_left[node_id], tree
                       .children_right[node_id])
18 }
```

V. Evaluation

A. Experimental Settings

We use the UNSW-NB15 dataset [33] that consists of 100GB pcap files from synthetic normal activities and contemporary attack behaviors (e.g., DoS and Exploits). Given a packet, the task of Dryad is to classify whether it is malicious

[4]https://jinja.palletsprojects.com/en/3.0.x/

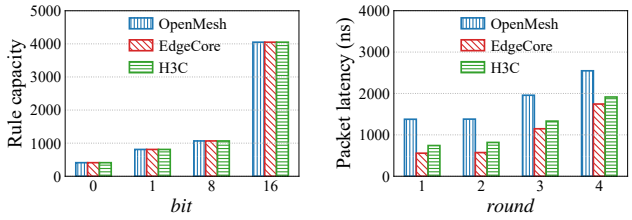
or not on switches. Learning from [9], [21], we consider the parsed IP/TCP/UDP header fields in the PHV as ODT features. The dataset is randomly splitted for training (80%) and testing (20%) on a server with Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz, Python 3, and scikit-learn [32]. Notably, by utilizing registers in switches to record flow statistics (e.g., the average and max packet sizes [19], [20]), Dryad can be used to identify whether a flow is malicious. Dryad's flow classification will be studied in future works.

We leverage three commodity switches, i.e., OpenMesh BF-48X6Z [25], EdgeCore Wedge 100BF-65X [26], and H3C S9850-32H [27] to demonstrate the performance of Dryad:

- For the OpenMesh switch, we constrain the max number of available stages to be 5, and the traffic rate is 10Gbps. In this case, the ODT accuracy is more important than the packet latency, so we set the progressive search coefficient $\alpha = 0.9$ in Algorithm 4.
- For the EdgeCore switch, the number of available stages is 8, the traffic rate is 40Gbps, and $\alpha = 0.7$.
- For the H3C switch, the number of stages is 12, the traffic rate is 100Gbps, and α is set to 0.5 to attach importance to the packet latency in such a high traffic rate case.

We set the max round $k = 4$, i.e., $round \in \{1, 2, 3, 4\}$ in the outermost loop of Algorithm 4: The Norm. and the Resub. solutions are indicated by $round = 1$ and $round = 2$, respectively, while $round \geq 3$ implies the Recir. solution.

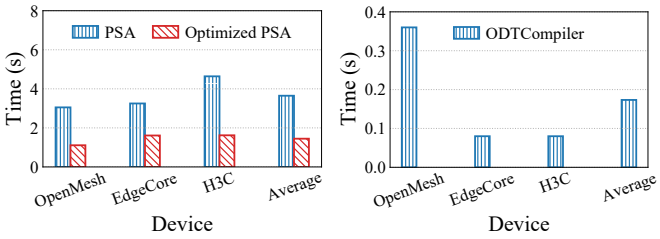
B. Rule and Latency Measurement



(a) Rule capacity (b) Packet process latency

Fig. 6: Rule capacity and packet process latency estimation of different switches.

Fig. 6 shows the rule capacity of different bit, and the packet process latency of different $round$ on three switches. For the rule capacity, we empirically measure the number of allowed table entries per stage when features with bits $\leq bit$ use the ternary match ($> bit$ use the range match). As shown in Fig. 6a, as more features are converted to the ternary match (i.e., bit from 0 to 16), the rule capacity increases from 400 to 4000. But as discussed in §IV-A2, using only the ternary match does not necessarily lead to better resource utilization, because threshold comparisons of features are natively ranges (e.g., $0 < f^* \leq f_t^*$), and it may consume $w\times$ table entries when one w-bit range is converted from the range match to the ternary match [38]. The results in Fig. 6b show that the packet process latency increases when the number of loopback rounds gets larger.



(a) Progressive search algorithm (PSA) (b) ODTCompiler

Fig. 7: The Dryad performance on different switches (PSA and ODTCompiler are based on Python3).

C. Adaptation Strategies and Their Effects

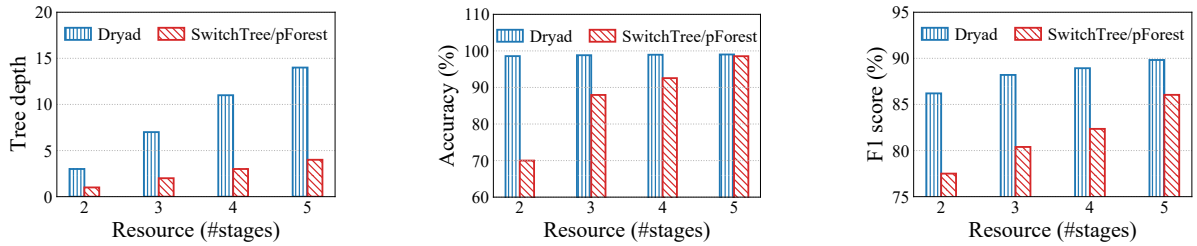
To show the real ODT deployment outcomes, we run Algorithm 4 on CPUs of each commodity switch and obtain the following adaptation strategies:

- For the OpenMesh switch (the case of 5 stages, 10Gbps), the algorithm finally selects the Recir. solution of $round = 4$ where features with bits ≤ 8 use the ternary match while the others use the range match. As packets are processed by the pipeline four times ($round = 4$), we ideally can obtain a pruned ODT of $4 \times 4 = 16$ levels (the fifth stage is used for recirculating, see §IV-A1). However, due to the rule capacity, we actually prune the ODT to a depth of 14 (15 levels).
- For the EdgeCore switch (the case of 8 stages, 40Gbps), the Resub. solution ($round = 2$) is selected where features with bits ≤ 8 use the ternary match. As packets are processed by the pipeline twice ($round = 2$), we obtain a pruned ODT of depth 13 ($2 \times 7 = 14$ levels, the eighth stage is used for resubmitting).
- For the H3C switch (the case of 12 stages, 100Gbps), the Norm. solution is selected with all features using the ternary match, and the ODT depth is 11 (12 levels).

Fig. 7a and 7b illustrate the time consumption of searching and deploying the aforementioned strategies in different switches. Notably, the progressive search algorithm (PSA) with and without optimizations in §IV-B2 are also included. On average, PSA and optimized PSA cost $3.64s$ and $1.44s$ respectively. The ODTCompiler that adjusts the ODT by pruning and generates P4 codes is much faster, and only costs $0.17s$ on average. That is, on average, it takes $1.61s$ to finish the ODT adaptation on a tested switch.

D. Comparison with Baselines

As stated in §IV-A1, SwitchTree [19] and pForest [20] use the same level-table manner, which is similar to our Norm. solution. Therefore, their DT depths are also correlated to stages. However, these two approaches do not discuss how to adjust the DT when the number of stages varies. So we assume they use the off-the-shelf CART (a DT variant) retraining provided by scikit-learn [31], [32] to adjust their depths. For the sake of simplicity, we do not consider the rule capacity



(a) Tree depth (b) Accuracy (c) F1-score

Fig. 8: The tree adaption according to the OpenMesh settings in §V-A.

limitation in these methods. The adaptation comparison in this section is simulated on our high-performance Intel server.

In Fig. 8, we can see that Dryad outperforms SwitchTree/pForest for the OpenMesh switch. While tree depth is limited by the stages for SwitchTree/pForest, Dryad can support deeper trees on OpenMesh as shown in Fig. 8a by employing our P4 scaling operations (§IV-A). As a result, Dryad achieves better classification performance as shown in Fig. 8b and 8c. E.g., $0.47\% \uparrow$ (99.02% vs. 98.55%) on accuracy and $3.78\% \uparrow$ (89.82% vs. 86.04%) on F1-score with #stages = 5.

In Table I, we present the adaptation time for different switch settings. We can see that pruning is more efficient than training a tree from scratch (i.e., retraining). For example, under the H3C case, Dryad is $161\times$ faster when pruning than the retraining in SwitchTree/pForest $(0.11s$ vs. $17.71s)$. Also, Dryad has a slightly higher retraining time than SwitchTree/pForest. This is because Dryad generally produces a deeper tree during training since our P4 scaling operations allow Dryad to support deeper trees (see §V-C).

TABLE I: The comparison of adaption time (seconds).

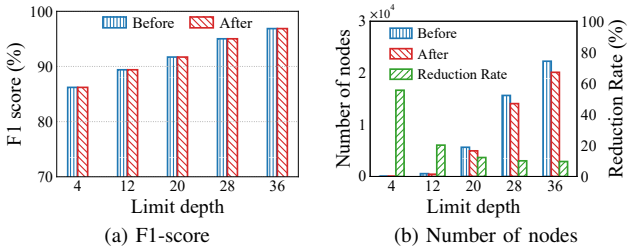
Switch	Dryad (Retraining)	Dryad (Pruning)	SwitchTree/pForest (Retraining)
OpenMesh (5 stages)	19.32	0.23	16.37
EdgeCore (8 stages)	18.54	0.18	16.70
H3C (12 stages)	17.83	0.11	17.71

E. Impact of Soft Pruning

Next, we investigate the impact of soft pruning. We see in Fig. 9a that soft pruning has no impact on ODT classification performance and in Fig. 9b that soft pruning can significantly reduce the number of nodes. In particular, for an ODT of depth 20, the F1-score remains the same after the soft pruning, but the number of nodes is reduced by 12.20%.

VI. Conclusion and Further Discussion

In this paper, we propose Dryad, a self-adaptive in-network intelligence system that can adapt to changing switch resources. In Dryad, a large and accurate ODT model is trained at maximum accuracy without any limits on the resulting depth of the DT. When there is a change in the available resources, we first run the progressive search algorithm to select the optimal ODT configuration. Then, we use the ODTCompiler to generate the corresponding P4 program. We show with extensive experiments that our Python-based prototype can finish the adaptation in $1.61s$. This is some $161\times$ faster than retraining afresh using the full dataset.



(a) F1-score (b) Number of nodes

Fig. 9: The F1-score and the number of nodes before/after the soft pruning. Reduction rate $= \frac{\text{Before}-\text{After}}{\text{Before}} \times 100\%$

Our current prototype is implemented in Python. The time required is still significantly longer than the time taken by commodity P4 switches to perform reconfiguration, which is typically in the order of nanoseconds [23]. We believe that this can be mitigated by re-implementing Dryad with a more efficient language like C++ that can be $25\times$ [40] to $29\times$ [28] faster than Python. Also, we can cache (or run offline) the frequently used configurations for acceleration.

VII. Acknowledgment

We thank Prof. Ben Leong for his helpful suggestions and the anonymous reviewers for their thoughtful comments. This work is supported in part by the National Key R&D Program of China under Grant No. 2022YFB3105000, the National Natural Science Foundation of China under Grant No. 61972189, the Major Key Project of PCL under Grant No. PCL2023AS5-1, the Shenzhen Key Lab of Software Defined Networking under Grant No. ZDSYS20140509172959989, and the China Scholarship Council (CSC202306210169).

REFERENCES

[1] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium, 2018*. The Internet Society, 2018.

[2] M. Amanowicz and D. Jankowski, "Detection and classification of malicious flows in software-defined networks using data mining techniques," *Sensors*, vol. 21, no. 9, p. 2972, 2021.

[3] R. Doshi, N. J. Apthorpe, and N. Feamster, "Machine learning ddos detection for consumer internet of things devices," in *Proceedings of the IEEE Security and Privacy Workshops (SPW)*. IEEE Computer Society, 2018, pp. 29–35.

[4] M. Lotfollahi, R. S. H. Zade, M. J. Siavoshani, and M. Saberian, "Deep packet: A novel approach for encrypted traffic classification using deep learning," *arXiv preprint arXiv:1709.02656*, 2017.

[5] R. Li, X. Xiao, S. Ni, H. Zheng, and S. Xia, "Byte segment neural network for network traffic classification," in *Proceedings of the 26th International Symposium on Quality of Service*. New York, USA: ACM, 2018, pp. 1–10.

[6] G. Xie, Q. Li, and Y. Jiang, "Self-attentive deep learning method for online traffic classification and its interpretability," *Computer Networks*, p. 108267, 2021.

[7] G. Xie, Q. Li, Y. Jiang, T. Dai, G. Shen, R. Li, R. Sinnott, and S. Xia, "Sam: Self-attention based deep learning method for online traffic classification," in *Proceedings of the Workshop on Network Meets AI & ML, NetAI@SIGCOMM*. ACM, 2020, pp. 14–20.

[8] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Y. Geng, L. Chen, K. Chen, and H. Jin, "Online flow size prediction for improved network routing," in *Proceedings of the 24th IEEE International Conference on Network Protocols*. IEEE Computer Society, 2016, pp. 1–6.

[9] Z. Xiong and N. Zilberman, "Do switches dream of machine learning?: Toward in-network classification," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. ACM, 2019, pp. 25–33.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[11] T. P. A. W. Group, *P4$_{16}$ Portable Switch Architecture (PSA)*, 2018 (accessed May 3, 2023). [Online]. Available: https://p4.org/p4-spec/docs/PSA-v1.0.0.html

[12] G. Xie, Q. Li, Y. Dong, G. Duan, Y. Jiang, and J. Duan, "Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation," in *Proceedings of the International Conference on Computer Communications*, 2022, pp. 1938–1947.

[13] I. Corporation, *Intel Tofino: P4-programmable Ethernet switch ASIC that delivers better performance at lower power*, 2019 (accessed May 3, 2023). [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html

[14] J. Xing, K. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen, "Runtime programmable switches," in *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2022, pp. 651–665.

[15] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[16] R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[17] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth, 1984.

[18] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Iisy: Practical in-network classification," *arXiv preprint arXiv:2205.08243*, 2022.

[19] J.-H. Lee and K. Singh, "Switchtree: in-network computing and traffic analyses with random forests," *Neural Computing and Applications*, pp. 1–12, 2020.

[20] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "pforest: In-network inference with random forests," *arXiv preprint arXiv:1909.05680*, 2019.

[21] C. Zheng and N. Zilberman, "Planter: seeding trees within switches," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, Poster and Demo Sessions*. ACM, 2021, pp. 12–14.

[22] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: adaptive and fast network-wide measurements," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 561–575.

[23] B. Antonin, *Leveraging stratum and tofino fast refresh for software upgrades*, 2018 (accessed August 3, 2023). [Online]. Available: https://opennetworking.org/wp-content/uploads/2018/12/Tofino_Fast_Refresh.pdf

[24] Y. Feng, Z. Chen, H. Song, W. Xu, J. Li, Z. Zhang, T. Yun, Y. Wan, and B. Liu, "Enabling in-situ programmability in network data plane: From architecture to language," in *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2022, pp. 635–649.

[25] T. Corporation, *OpenMesh BF-48X6Z Programmable Switch*, 2022 (accessed May 3, 2023). [Online]. Available: http://www.tooyum.com/products/OpenMesh_BF48X6Z.html

[26] E. N. Corporation, *DCS802 12.8T PROGRAMMABLE DATA CENTER SWITCH*, 2021 (accessed May 1, 2023). [Online]. Available: https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=334

[27] N. H. T. Corporation, *H3C S9850 Series Data Center Switches*, 2022 (accessed May 1, 2023). [Online]. Available: https://www.h3c.com/en/Products_and_Solutions/InterConnect/Switches/Products/Data_Center/Aggregation/S9800/H3C_S9850/

[28] D. Lion, A. Chiu, M. Stumm, and D. Yuan, "Investigating managed language runtime performance: Why javascript and python are 8x and 29x slower than c++, yet java and go can be faster?" in *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2022, pp. 835–852.

[29] P. Cui, H. Pan, Z. Li, J. Wu, S. Zhang, X. Yang, H. Guan, and G. Xie, "Netfc: Enabling accurate floating-point arithmetic on programmable switches," in *Proceedings of the 29th IEEE International Conference on Network Protocols*. IEEE, 2021, pp. 1–11.

[30] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, and A. Madeira, "Flowlens: Enabling efficient flow classification for ml-based network security applications," in *Proceedings of the 28th Annual Network and Distributed System Security Symposium*. The Internet Society, 2021.

[31] scikit learn, *Decision Trees*, 2007 (accessed August 3, 2023). [Online]. Available: https://scikit-learn.org/stable/modules/tree.html#tree

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

[33] N. Moustafa and J. Slay, "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)," in *Proceedings of the Military Communications and Information Systems Conference*. IEEE, 2015, pp. 1–6.

[34] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," in *Proceedings of the 57th Conference of the Association for Computational Linguistics*. Association for Computational Linguistics, 2019, pp. 3645–3650.

[35] L. A. Breslow and D. W. Aha, "Simplifying decision trees: A survey," *Knowledge Engineering Review*, vol. 12, no. 1, pp. 1–40, 1997.

[36] J. R. Quinlan, "Simplifying decision trees," *International Journal of Human-Computer Studies*, vol. 51, no. 2, pp. 497–510, 1999.

[37] M. Bohanec and I. Bratko, "Trading accuracy for simplicity in decision trees," *Machine Learning*, vol. 15, no. 3, pp. 223–250, 1994.

[38] H. Liu, "Efficient mapping of range classifier into ternary-cam," in *Proceedings of the 10th Annual IEEE Symposium on High Performance Interconnects*. IEEE Computer Society, 2002, pp. 95–100.

[39] P. S. Foundation, *Process-based parallelism*, 2022 (accessed April 3, 2023). [Online]. Available: https://docs.python.org/3/library/multiprocessing.html

[40] N. Tamimi, *How Fast Is C++ Compared to Python?*, 2020 (accessed August 3, 2023). [Online]. Available: https://towardsdatascience.com/how-fast-is-c-compared-to-python-978f18f474c7