frontiers

ML-NIC: Accelerating Machine Learning Inference using Smart Network Interface Cards

Raghav Kapoor [1], **David C. Anastasiu** [2] **and Sean Choi** [1*]

[1] *Cloud Lab, Department of Computer Science and Engineering, Santa Clara University, Santa Clara, CA, United States*
[2] *Anastasiu Lab, Department of Computer Science and Engineering, Santa Clara University, Santa Clara, CA, United States*

Correspondence*:
Sean Choi, David C. Anastasiu
sean.choi@scu.edu, danastasiu@scu.edu

ABSTRACT

Low-latency inference for machine learning models is increasingly becoming a necessary requirement, as these models are used in mission-critical applications such as autonomous driving, military defense (e.g., target recognition), and network traffic analysis. A widely studied and used technique to overcome this challenge is to offload some or all parts of the inference tasks onto specialized hardware such as graphic processing units. More recently, offloading machine learning inference onto programmable network devices, such as programmable network interface cards or a programmable switch, is gaining interest from both industry and academia, especially due to the latency reduction and computational benefits of performing inference directly on the data plane where the network packets are processed. Yet, current approaches are relatively limited in scope, and there is a need to develop more general approaches for mapping offloading machine learning models onto programmable network devices.

To fulfill such a need, this work introduces a novel framework, called ML-NIC, for deploying trained machine learning models onto programmable network devices' data planes. ML-NIC deploys models directly into the computational cores of the devices to efficiently leverage the inherent parallelism capabilities of network devices, thus providing huge latency and throughput gains. Our experiments show that ML-NIC reduced inference latency by at least $6\times$ on average and in the 99th percentile and increased throughput by at least $16x$ with little to no degradation in model effectiveness compared to the existing CPU solutions. In addition, ML-NIC can provide tighter guaranteed latency bounds in the presence of other network traffic with shorter tail latencies. Furthermore, ML-NIC reduces CPU and host server RAM utilization by 6.65% and 320.80 MB. Finally, ML-NIC can handle machine learning models that are $2.25\times$ larger than the current state-of-the-art network device offloading approaches.

Keywords: Machine Learning, SmartNIC, Netronome, Data Plane, Inference

1 INTRODUCTION

Machine learning (ML) permeates a vast amount of everyday life, from personalized recommendations to stock market analysis and novel drug synthesis. While the machine learning models created to solve problems in these various fields are proven to be highly effective, these models often need large amount

of time to make predictions (also referred to as model inference) on data instances. Often times, such limitation becomes a huge limiting factor for deploying ML models for latency critical applications. For example, applications such as high-frequency trading, military target recognition, pilots traveling at aircraft speeds (i.e., at least 621 mph) need to perform ML inference with the tight latency budget of at most 50 ms. Making things worse, it is often not computationally and physically feasible to host large, effective ML models on directly on the devices like military aircraft. Thus, the ML model computations are often offloaded onto ground stations or edge devices, where data transmission can significantly add to the latency to execute model inference.

While many different types of machine learning accelerators have been developed, such as Graphical Processing Units (GPU) (Choquette et al., 2018), Field Programmable Gate Arrays (FPGA) (Fowers et al., 2018; He et al., 2018; Tong et al., 2017), and specialized application-specific integrated circuits (ASIC) (Chen et al., 2014; Jouppi et al., 2017), their efficiency is lowered by the data transfer time over the PCIe bus from the host system's network interface card (NIC). To overcome this challenge, we investigated methods to perform ML inference at the edge of the network to reduce the need and the overhead of transferring data from the edge to these accelerators. To achieve this, we note that the emergence of programmable data planes makes network devices (i.e., programmable switches and NICs) potential candidates for accelerating ML inference, especially given that programmable network devices have already been shown to be significantly power efficient while also providing high throughput and low latency in a variety of in-network computing tasks such as caching (Jin et al., 2017), consensus (Dang et al., 2020), and network monitoring (Kim et al., 2015). However, leveraging programmable data planes for machine learning inference still is an ongoing area of research with room for improvement.

Much of the prior works in this area has shown that network devices with programmable data planes, primarily programmable switches, demonstrate superior latency performance with minor degradation in model effectiveness (Zhang et al., 2023). While the line rate performance of programmable switches is beneficial for model inference, their limitation to match+action logic, memory size, cost and the placement in the network restrict the feasibility and accuracy of models that can be mapped onto them. For example, since many modern machine learning algorithms rely on operations such as multiplication during inference, finite-sized match+action tables cannot support every possible combination of multiplied values. Even though prior methods have found ways around this, it was not without loss in model effectiveness. And as machine learning models continue to grow, additional losses in model effectiveness seem likely. In addition, prior methods focused on general models that may not be used in the real-world for low-latency applications.

To compensate for this limitation, we propose the Smart Network Interface Cards (SmartNICs) as a viable alternative. SmartNICs possess additional computational resources and several packet processing accelerators that can be adapted to mimic essential machine learning inference operations, such as multiplication and logarithm functions, more accurately. Furthermore, SmartNICs are much more cost and power efficient, are more easier to deploy and test.

Therefore, in this paper, we present ML-NIC, a framework for compiling and deploying trained machine learning models onto SmartNICs by providing intelligent model mapping methods. This current work mainly focuses on mapping tree-based models onto SmartNICs due to it's wide usage of low-latency applications, but we have proposals for future work with proposed methods to support inference for other types machine learning models as well. Compared to many prior works that implement machine learning algorithms onto programmable network devices, ML-NIC implementation uses more device parallelism in

the inference process. Finally, our Python implementation of ML-NIC is made publicly available upon publication of the manuscript.

Our contributions include:

1. We present an algorithm to extract logic learned by a generic decision tree to facilitate parallelized feature analysis during inference.
2. We present a method to map and compile trained decision trees onto a SmartNIC in a manner that leverages its parallelism capabilities.
3. We demonstrate our framework's potential for accelerating the inference of decision trees for different tasks compared to conventional CPU and current state-of-the-art SmartNIC model deployment strategies.
4. We created an open-source project that contains all of ML-NIC's implementation and experimentation [1].

The rest of this paper is organized as follows. Section 2 presents some background information on SmartNICs relevant to our work. Section 3 explains our approach towards deploying machine learning models onto a SmartNIC. Sections 4, 5, and 6 describe our experimental setup and discuss our results. Section 7 presents an overview of recent work that utilizes programmable data planes to accelerate the inference time for various machine learning algorithms on different problems. Section 8 points out future directions, and Section 9 ends with concluding remarks.

2 BACKGROUND

In this section, we provide the background for ML-NIC and some underlying motivations.

2.1 SmartNIC

Smart Network Interface Cards (SmartNICs) possess additional computational resources and memory storage compared to traditional network interface cards. These resources enable SmartNICs to perform deep packet inspection, network function virtualization, and zero-trust security (Netronome Systems, 2024). As a result, offloading such operations to the SmartNIC frees a host system's CPU from conducting them. Compared to programmable switches, SmartNICs have more computational resources that can be leveraged. This motivates our choice to use SmartNICs for machine learning inference, since this process can be quite computationally intensive. For the rest of the section, we will focus on one particular type of SmartNIC: ASIC-Based Netronome SmartNICs supporting the NFP4000 architecture.

2.1.1 ASIC-Based Netronome SmartNIC

ASIC-based SmartNIC represents a type of SmartNIC where the ASIC is custom built to support programmability using a set of custom languages. One notable example of an ASIC-based SmartNIC is a Netromome SmartNIC that can be programmed using langauged called P4 and Micro-C. This project utilizes a specific subset of the Netromome SmartNICs, which support the NFP4000 architecture. To elaborate further, Netronome SmartNICs support the NFP4000 architecture feature 48 packet processing cores and 60 programmable flow processing cores (Corigine, 2020). In much of the literature and technical documentation, the flow-processing cores are referred to as microengines (ME), which we denote as purple squares in Figure 1. Each microengine acts as an independent 32-bit processor with its own code store and local memory to run different programs in parallel with the other microengines. The microengines can be

[1] The project can be found on https://github.com/The-Cloud-Lab/ML-NIC

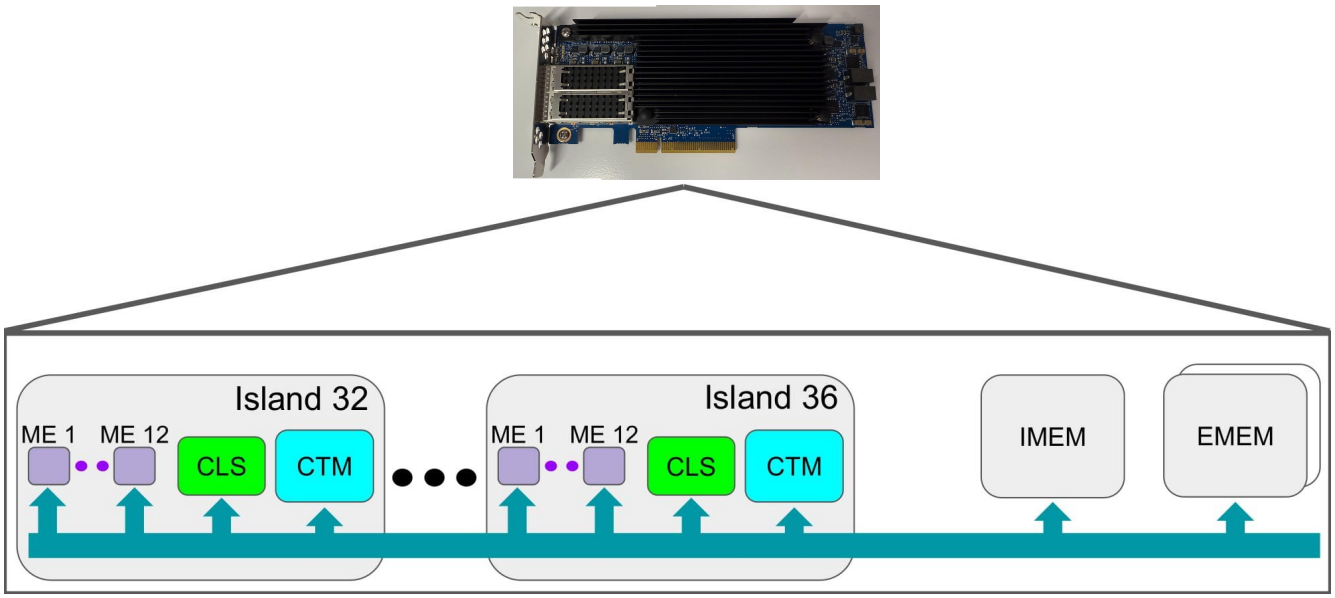

Figure 1. In this figure, we present a high-level overview of the NFP4000 architecture, focusing on the components most pertinent to ML-NIC.

programmed using a low-level language like Micro-C, an extended subset of C89, or a high-level language like P4. A key difference between Micro-C and P4 is that Micro-C provides the flexibility to program each microengine differently, whereas P4 defaults to loading the same program onto all microengines. However, both languages lack floating-point number support. We provide a high-level illustration of the NFP4000 architecture in Figure 1.

Each microengine supports 8 threads, where each thread runs the same program and has its own block of memory/registers. The following memory in a microengine is evenly partitioned among the 8 threads in a microengine:

- 256, 32-bit General-Purpose Registers - used for general per-packet computations
- 256, 32-bit Transfer Registers - used for transferring data between memory regions
- 128, 32-bit Next-Neighbor Registers - used for communicating between neighboring microengines in the same island
- 4kB of Local Memory - used for additional data storage as needed
- 120 Signal Registers - used to notify threads that a certain hardware event has occurred

The partitioning of memory among the 8 threads facilitates fast context switching between them, so they can process different packets efficiently (Siracusano et al., 2022).

The microengines are organized into islands. While these islands can vary in number of microengines and specialized functionality, standard islands, shown in Figure 1, contain 12 microengines with two regions of memory shared between all the microengines in the island: Cluster Local Scratch (CLS) and Cluster Target Memory (CTM). CLS, denoted in green in Figure 1, commonly stores small forwarding tables shared between the microengines (Wray, 2014). CTM, denoted in cyan in Figure 1, holds packet headers and coordinates between the microengines and other subsystems on the card (Wray, 2014). As CTM is larger than CLS, more clock cycles are required to read/write to CTM.

Outside of the islands, the Netronome SmartNICs have three additional memory units, as shown in Figure 1, shared with all microengines: one Internal Memory Unit (IMEM) and two External Memory Units (EMEM) (Langlet, 2019). IMEM is used for storing packet payloads and medium-sized match-action tables (Wray, 2014). EMEM is used to store larger match-action tables and other flow statistics (Wray, 2014). As these three memory units are the largest of those mentioned prior, with EMEM being larger than IMEM, they require a greater number of clock cycles to read/write to them. Microengines can access data in all these memory regions using the Command Push Pull (CPP) bus, denoted in turquoise in Figure 1.

Based on our knowledge of data access times for the different memory regions, hardware signals, and transfer registers, we design ML-NIC to efficiently leverage these resources to ensure a high degree of performance from a SmartNIC.

3 ML-NIC ARCHITECTURE

ML-NIC comprises three components: machine learning model training, model mapping, and model deployment, as shown in Figure 2. We explain the details of each component below.

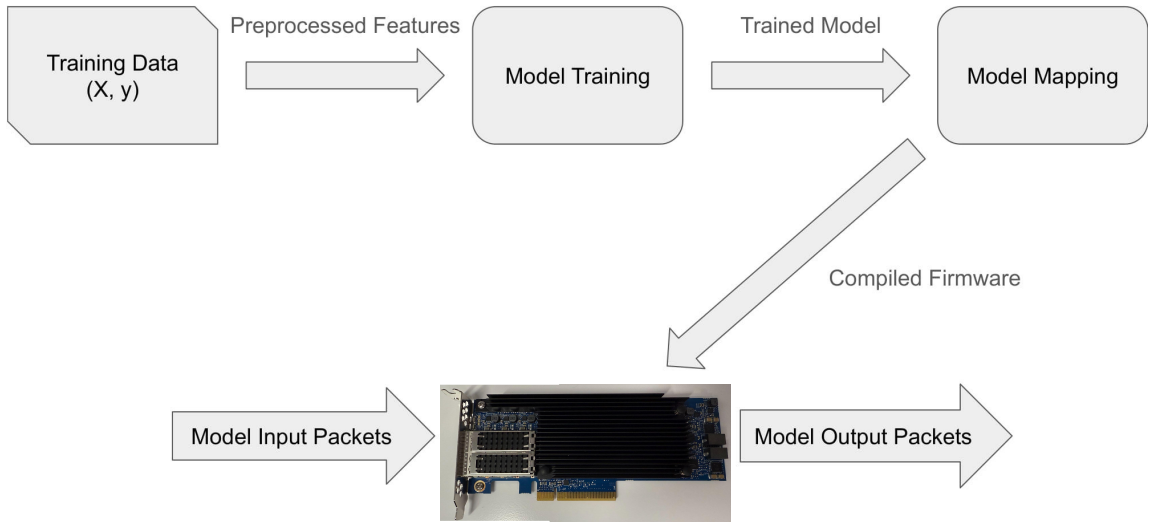

Figure 2. This figure shows the big picture of the ML-NIC architecture.

3.1 Model Training

In the machine learning model training component, we consider a labeled dataset $(X,\ y)$, where $X \in \mathbb{R}^{mxn}$ represents our data matrix (m data points, n features) and $y \in \{1 \dots q\}^m$ represents our class labels (q possible classes). We make no assumptions on whether X consists of only continuous features, only categorical features, or a mix. We assume the continuous features are normalized within the range $[0,\ 1]$. We do so to simplify the range of numerical representation that the first iteration of ML-NIC needs to account for. We find that this assumption is reasonable, since data normalization is a common technique in machine learning to prevent certain features from dominating over other features due to differences in scaling. For the categorical features, we assume that they are one-hot encoded (Liu, 2017) (i.e., a feature with three categories is expanded to three features with values 0 or 1). This dataset is used to learn the parameters of a particular machine learning model.

In this first iteration of ML-NIC, we choose to focus on decision trees for three reasons. First, we cite the relative computational simplicity of tree traversal compared to floating-point operations in hardware without a Floating-Point Unit (FPU) like a SmartNIC that was designed for fast computations at network line rates.

Second, in comparison to other classical supervised machine learning models, such as Naive Bayes, k-nearest neighbor, and support vector machine, we find that the decision tree is the more suitable choice for offloading. With Naive Bayes, it is known that the algorithm perform poorly when the features used for training are not conditionally independent. However, in practice, decision trees can perform well even if the features are correlated. For k-nearest neighbor, we find that the required storage of every training instance to be an obstacle for offloading onto network devices, especially given the size of modern datasets. Even if only a selection of the training set was used to make offloading feasible, this could result in more significant performance degradation in certain machine learning problems. With respect to support vector machines for multiclass classification, the model may not be optimal for offloading with a large number of features and number of classes. For this explanation, we temporarily denote m to be the number of features, q as the number of classes (greater than 2 for multiclass classification), and z as the number of support vectors. First, assume the support vectors for the support vector machine can be stored and there exists suitable means of multiplication and a kernel on an off-the-shelf programmable network device that are at least as expensive as a comparison operation. Since off-the-shelf programmable network device are optimized for match + action, we expect the device architecture to have an efficient compare operation. Then, we see that the support vector machine requires at least $m \times z \times q$ multiplications, whereas a decision tree would require at most m compare operations for a single inference. Therefore, based on instruction count, the decision tree model has a better chance of yielding inference latency reduction when offloaded onto an off-the-shelf programmable network device.

Third, we note that tree-like machine learning models, such as XGBoost (Chen and Guestrin, 2016), are commonly used to learn tasks from structured tabular data over neural networks given faster training time, potential performance gain, and model transparency. We use the decision tree model to show that our framework can be used in real-world applications and offloading a machine learning model onto a SmartNIC can reduce model inference latency significantly. While our focus is currently on decision tree inference, we provide a discussion on how our framework can be augmented to account for additional machine learning models in Section 6.

To train a decision tree, we consider the high-level algorithm outlined in Algorithm 1. Since finding the globally-optimal tree structure for a learning task is computationally challenging, locally-optimal heuristic algorithms are used such as ID3 (Quinlan, 1986), C4.5 (Quinlan, 1986), and CART (Leo Breiman and Olshen, 1984). In practice, metric M is commonly Information Gain in the case of ID3 and C4.5 or Gini Impurity for CART. Formulations for these metrics are provided in Equations 1 and 2. In Equations 1 and 2, we further define C as the number of classes, P as the number of splits, S_c as the number of examples in the training dataset subset with class label c, and $S_{p,c}$ as the number of examples in the pth split of the training dataset subset with class label c.

$$Gini(S) = \sum_{c=1}^{C} \frac{|S_c|}{|S|} \left(1 - \frac{|S_c|}{|S|}\right) \tag{1}$$

Algorithm 1 Decision Tree Training Algorithm

Require: S is a valid subset of the training dataset D, R is a set of stopping criteria for the algorithm, M is a valid impurity metric to locally optimize, $split\_node$ is a valid function for a splitting S at a node

function TRAIN_TREE($S, R, split\_node$)
  $c = majority\_label(S)$
  $tree\_node = Node(label = c)$
  if $not\_satisfied(R)$ **then**
    $ms = list()$
    $splits = split\_node(S)$
    for each $split \in splits$ **do**
      $ms.append(M(split))$
    end for
    $best\_split = splits[arg\_optimal(ms, M)]$
    for each $s \in best\_split$ **do**
      $tree_node.insert_branch($TRAIN_TREE($s, R, split\_node$))
    end for
  end if
  `return` $tree\_node$
end function

$$Info(S) = \sum_{c=1}^{C} \left(-\frac{|S_c|}{|S|} \log_2 \left(\frac{|S_c|}{|S|} \right) \right) - \sum_{p=1}^{P} \frac{|S_p|}{|S|} \sum_{c=1}^{C} \left(-\frac{|S_{p,c}|}{|S_p|} \log_2 \left(\frac{|S_{p,c}|}{|S_p|} \right) \right) \quad (2)$$

3.2 Model Mapping

Before discussing the technical details of decision tree mapping onto a SmartNIC, we discuss our mapping approach at a high-level. To run inference, we find the disjunctive normal form (Roth, 2016) of a decision tree. In the disjunctive normal form, the logic for assigning a class label to a data instance is expressed as a disjunction of conjunctions (i.e., (condition 1 and condition 2 and ...) or (condition3 and condition1 and ...) or ...). Each conjunction in the disjunction (i.e., condition 1 and condition 2 and ...) represents a path from the root node to a leaf node in a decision tree. We prefer the disjunctive norm form over the typical tree structure of a decision tree for inference, since it makes executing inference in a parallelized manner more convenient. To parallelize the inference process from the disjunctive normal form, we take the conditions from all the conjunctions that correspond to a particular feature, noting which path in the decision tree the condition corresponds to. To run inference on a data instance then, the conditions for each feature can be evaluated in parallel, where the result of each feature evaluation yields a set of paths in the decision tree that are possible for the data instance to take. Then, by aggregating the all possible paths and taking the intersection among them, a single path can be found. By matching the path to its corresponding class label, the decision tree inference process is complete.

To map a decision tree onto a SmartNIC, we take the output of the machine learning model training process (i.e., a pickle file) and proceed to generate an implementation of the SmartNIC data plane. Currently, we support SmartNICs that are programmable in Micro-C, primarily SoC-based Netronome SmartNICs. In the current iteration of our framework, we consider a trained decision tree classifier C with l leaf nodes, where l is at most 256. We make no additional assumptions on the number of splits per non-leaf node or the training algorithm used. Based on the number of leaf nodes l in model C and number of features n in X, there are three possible scenarios for mapping C onto the SmartNIC:

- The model can fit on one island of the SmartNIC. Each island is then programmed with its own set of feature computation, result aggregation, and packet collection microengines (i.e., inference for model C is run on all the islands)
- The model can fit on the entire SmartNIC with one feature assigned per feature microengine and one packet collection microengine.
- The model can fit on the entire SmartNIC with multiple features assigned per feature microengine and one packet collection microengine.

After selecting one of the three above mapping schemes, the next step is to extract the logic (i.e., find disjunctive normal form and extract the conditions that match to a particular feature) learned by model C. To do so, we iterate through all the n features in X and perform a depth-first search through the decision tree. We record the operation for those nodes that run a comparison operation on our feature of interest and continue the depth-first search until all the leaf nodes have been reached. Formally, Algorithm 2 illustrates our logic extraction approach.

Algorithm 2 Decision Tree Logic Extraction Algorithm

Require: $node$ points to valid node in decision tree, $ftre$ is feature seen by decision tree during training, clt has enough space to store decision tree logic for $ftre$

```
1:  function GET_LOGIC(node, ftre, clt)
2:      if is_leaf(node) then
3:          clt.insert(node.prediction)
4:      else
5:          if node.ftre = ftre then
6:              clt.insert(node.logic)
7:          end if
8:          for each child ∈ node.children do
9:              clt.insert(GET_LOGIC(child, ftre, clt))
10:         end for
11:     end if
12:     return clt
13: end function
```

We also assign the each of microengines on the SmartNIC as one of three types: packet collection, feature computation, and result aggregation. The packet collection microengine(s) are programmed to signal the CTM packet engine that they are ready to receive packets. Once a packet is received, the packet collection microengine(s) will verify that packet is a model input packet, extract the features from the packet payload, and asynchronously signal all the feature computation microengines of the inference request and transmit the corresponding feature to each via transfer registers.

The feature computation microengines are responsible for evaluating the conditions on a feature for a given data instance and determine which paths in the decision tree are possible. Since each feature computation microengine is responsible for different features and run simulatenously, all the features can be evaluated and all the possible paths in the decision tree can be determined in parallel. To implement the conditions and determine the possible paths per feature on the SmartNIC, we use Micro-C if-statements to evaluate the conditions and update an array of integers to reflect which paths are possible. For the update, we treat the array of integers as a single bit string, where most significant bit in the integer at the last index in the array corresponds to path 1. We assign paths based on the order in which the nodes are

encountered by Algorithm 2. Since the values for comparison in the conditions for evaluating each feature in the decision tree and the features themselves can be floating-point, and the SmartNIC does have an FPU, we consider an alternative floating-point representation. We represent floating-point numbers on the SmartNIC using a fixed-point representation that consists of 16 bits, where the last 13 bits represent the non-integer portion of a floating-point number. As each feature computation microengine completes its evaluation of its correponding feature, they notify the result aggregation microengine(s) of the decision tree paths that are possible based on the feature they each evaluation.

Once all the feature computation microengines finish their evaluation, the result aggregation microengine(s) finds the intersecting path the decision tree between all the possible paths, matches the path to the corresponding class label, and sends an asynchronous signal to the packet collection microengine(s) along with the class label via transfer register(s). Once the packet collection microengine(s) receives the signal from the result aggregation microengine(s), it edits the original packet payload with the class label for the data instance and notified the CTM packet engine that the packet needs to be transmitted. Also note that during the time the feature computation and result aggregation microengines are completing their tasks, the packet collection microengine(s) are editing the model input packet's header in preparation for transmission as a model output packet.

To program all the packet collection, feature computation, and result aggregation microengines, separate Micro-C code is written to program each microengine to complete their specific task for model inference, whereas prior methods often program all the microengines with one piece of P4 code to perform the same tasks for the model inference and do not fully leverage the parallel operating capacity of the SmartNIC. Example Micro-C code for packet collection, feature computation, and result aggregation can be found below in Listings 1, 2, and 3.

3.3 Model Deployment

Once all the Micro-C code files are created, they are all compiled and linked to generate the device firmware to run on the data plane in the model deployment component. Then, the firmware file output is loaded onto the SmartNIC. An example of the full process is shown in Listing 4. Each microengine assumes a specified behavior based on one of the three microengine assignments specified above. The SmartNIC can now ingress packets with features in the packet payload, run machine learning inference in a parallelized manner, and egress packets with the classification result as the packet payload.

4 EXPERIMENTAL SETUP

4.1 Testbed

Our testbed consists of two Dell PowerEdge Rack Servers. Server 1 hosts an NVIDIA Mellanox Bluefield-2 DPU 25 GbE SmartNIC for packet transmission and data collection. Server 2 hosts a Netronome AgilioCX 2×25 GbE SmartNIC, on which our decision tree models are deployed. Both systems are directly connected via qsfp cable between the Mellanox and Netronome SmartNICs. We illustrate our setup in Figure 3.

4.2 Datasets and Models

Our evaluation considers four tasks: land mine detection, satellite image pixel classification, gas sensor drift compensation, and network traffic classification. The main characteristics of the datasets used for each task can be found in Table 1. We train a decision tree model for each task using the scikit-learn

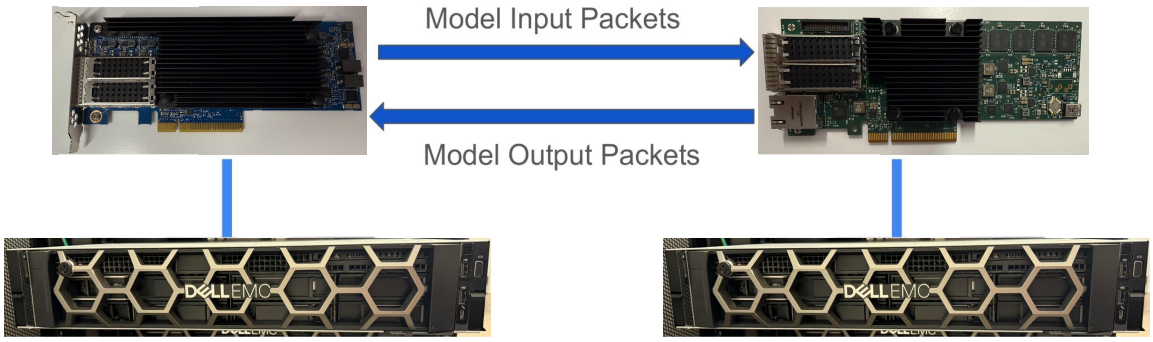

Figure 3. The testbed setup used for evaluation. The left server hosts the Netronome AgilioCX 2×25 GbE. The right server hosts the NVIDIA Mellanox Bluefield-2 DPU 25 GbE.

library (Pedregosa et al., 2011). We summarize the hyperparameters used for each tree in Table 2. For hyperparameters not explicitly mentioned in the table that can be tuned for the decision tree models (i.e., criterion, splitter, max features, etc.), we resort to the default values provided by scikit-learn.

Table 1. Summary of Datasets Used (refer to dataset subsections for class label abbreviations)

Attribute	Mine	Landsat	Gas	CICIDS
# of features	3	36	128	7
# of data instances	338	6435	13910	22887218
# of classes	5	6	6	7
# of training data	270	4435	11128	500000
# of test data	68	2000	2782	6957375

Table 2. Summary of decision tree Models Created

Parameter	Mine	Landsat	Gas	CICIDS
# of leaves	114	256	256	89
depth	17	15	24	15
# of nodes	227	511	511	177
min samples leaf	1	1	1	2
min samples split	2	2	2	2
min impurity decrease	0	0	0	0.00001
max leaf nodes	None	256	256	None

4.2.1 Dataset Preprocessing

As mentioned in Section 3, we assume the continuous features are in the range $[0, 1]$ and categorical features are one-hot-encoded. To achieve this, we apply min-max normalization to scale the continuous features of each dataset to range between 0 and 1 using the training set. Test features that lie outside the range $[0, 1]$ after min-max normalization has been applied are clipped to the closest endpoint. We also one-hot-encode the categorical features for each dataset based on the values observed from the training set. If the categorical features in the test set take on values not observed in the training set, the one-hot-encoded feature is represented as a bit string of zeros.

4.2.2 Land Mine Detection

We use the Land Mines dataset (Yilmaz et al., 2018) for the land mine detection task. The authors propose three features to classify a mine into five types, Null, Anti-Tank, Anti-Personnel, Booby-Trapped Anti-Personnel, and M14 Anti-Personnel, with $65 - 71$ samples per class. Our motivation for choosing this dataset is based on the number of features (8 after data preprocessing), where we can evaluate the first SmartNIC mapping scenario (fitting on one island) as described in Section 3. In later sections, we will refer to this dataset as Mine.

4.2.3 Satellite Image Pixel Classification

We use the Statlog (Landsat Satellite) dataset (Srinivasan, 1993) for the satellite image pixel classification task. The goal of this task is to examine multispectral values from a $3x3$ neighborhood of a satellite image and classify the central pixel as one of five classes: Red Soil, Cotton Crop, Grey Soil, Damp Grey Soil, Soil with Vegetation Stubble, Mixture, or Very Damp Grey Soil. There are $626 - 1533$ samples per class. Our motivation for choosing this dataset is based on the number of features (36), where we can evaluate the second mapping scenario (fitting on whole SmartNIC, one feature per microengine) as described in Section 3. In later sections, we will refer to this dataset as Landsat.

4.2.4 Gas Sensor Drift Compensation

We use the Gas Sensor Array Drift dataset (Rodríguez-Luján et al., 2014) for the gas sensor drift compensation tasks. This dataset consists of measurements from 16 chemical sensors to identify six gases, Ammonia, Acetaldehyde, Acetone, Ethylene, Ethanol, and Toluene, with $1508 - 3009$ samples per class. Our motivation for choosing this dataset is based on the number of features (128), where we can evaluate the third mapping scenario (fitting on whole SmartNIC, multiple features per microengine) as described in Section 3. In later sections, we will refer to this dataset as Gas.

4.2.5 Network Traffic Classification

We use the CICIDS2017 dataset (Sharafaldin et al., 2018) for the network traffic classification use case. This use case's purpose is to identify network flows as benign or malicious (brute force attack, heartbleed attack, botnet, DoS attack, DDoS attack, web attack, infiltration attack). However, we follow the approach used by Xavier et al. (2021) to generate the dataset for classifying individual network packets rather than network flows and the training and tests sets based on the network flows instead of the conventional stratified $80/20$ split used for the above datasets above. In the dataset, the packets were labeled as benign, DoS GoldenEye, DoS Hulk, DoS Slowhttptest, DoS Slowloris, Web Brute Force, or Port Scan. Each class has $30059 - 20121944$ samples. Our motivation for choosing this dataset is based on its use in the work by Xavier et al. (2021), which is similar to our approach. Our evaluation on this dataset clearly compares our approach and Xavier et al. (2021)'s approach. In later sections, we will refer to this dataset as CICIDS.

4.3 Baselines

We compare our approach against the following two baselines. First, we implement a traditional CPU baseline, which uses socket programming to receive incoming packets, extract the payload, run inference with the trained scikit-learn decision tree, and build and send a model output packet with the model prediction in the packet payload.

Second, we implement the approach developed by Xavier et al. (2021) using P4-16. Like our approach, Xavier et al. (2021)'s approach traverses through the scikit-learn decision tree structure, extracts the model's

logic, and rebuilds the tree in P4 using Python. Note that the original implementation was solely created for the CICIDS dataset, and the authors did not provide a method to handle floating-point features. In evaluating this method on the other datasets, we modified it slightly to use our fixed-point representation of floating-point numbers. Also, due to limitations with Xavier et al. (2021)'s approach, we could not evaluate it on larger decision trees, such as those generated with the Landsat and Gas Datasets.

4.4 Evaluation Metrics

In our experiments, we measure the effectiveness, (average and tail) latency, throughput, and hardware utilization of ML-NIC against the baselines.

For effectiveness, we measured the accuracy, F1 score, recall, and precision metrics on each dataset's test set. Given that our datasets are for multiclass classification tasks, we take the macro-average (i.e., unweighted mean) of the per-class scores for the F1 score, recall, and precision measurements.

For latency, we collected the time between model input packet transmission and model output packet reception on server 1 in microseconds for 1000 packets. In addition to our vanilla latency experiments (i.e., no CPU load or network link utilization), we conduct latency experiments with background traffic on the network link and CPU load. We generate random network traffic at different speeds using Tcpreplay for latency experiments with background traffic to achieve 25%, 50%, and 99% network link utilization. We use stress-ng for latency experiments with CPU load and generate CPU loads of 25%, 50%, and 99%. To ensure an apples-to-apples comparison with the CPU baseline, we append zero padding to the model input packets for our approach and the P4 baseline. Hence, they are the same size as the CPU model input packets. Note that, when collecting the data for the CPU baseline, we remove the time taken to decode the data features and encode the model's prediction.

For throughput, we use Tcpreplay to loop through the PCAP files containing the test set packets for each dataset at top speed. Simultaneously, we also run Tshark to filter and collect the model prediction packets for 60 seconds.

Lastly, for hardware utilization, we run Tcpreplay for 60 seconds like we did for the throughput experiment and measure CPU, server host RAM, and SmartNIC memory utilization. We also measure CPU, server host RAM, and SmartNIC memory utilization 30 seconds before and after running Tcpreplay for reference. We do not explicitly measure SmartNIC microengine utilization in this experiment. Instead, we use the results from our network link utilization and CPU load latency experiments as a proxy for SmartNIC microengine utilization.

5 RESULTS

From our experimental results, we demonstrate the following:

1. Our approach achieves effectiveness scores similar to those of the CPU baseline and identical to the P4 baseline.
2. Our approach has better a latency guarantee than the CPU and P4 baselines in various network link utilization and CPU load scenarios.
3. The throughput of our approach is significantly greater compared to the CPU baseline and on par with the P4 baseline.
4. Our approach uses fewer server host resources (i.e., CPU and server RAM) compared to the CPU and P4 baselines.

5.1 Effectiveness Scores

As shown in Table 3 [2], the effectiveness scores between our approach and the CPU baselines are similar with minor degradation. For conciseness, we only show the plots of accuracy and F1 score of the models, since the precision and recall results follow a similar pattern.

Table 3. Effectiveness Measurements On All Datasets

Dataset	Measure	CPU	Xavier et al. (2021)	ML-NIC
Mine	Accuracy (%)	**58.82**	57.35	57.35
	F1 Score (%)	**58.22**	56.92	56.92
Landsat	Accuracy	**85.65**	N/A	85.55
	F1 Score (%)	**83.93**	N/A	83.76
Gas	Accuracy (%)	**97.41**	N/A	95.15
	F1 Score (%)	**97.26**	N/A	94.78
CICIDS	Accuracy (%)	**95.12**	**95.12**	**95.12**
	F1 Score (%)	**47.04**	**47.04**	**47.04**

For the Mine dataset, we note differences of 1.47%, 1.30%, 1.35%, and 1.54% across the accuracy, F1 score, precision, and recall metrics. For the Landsat dataset, we note differences of 0.100%, 0.17%, 0.14%, and 0.20% across the accuracy, F1 score, precision, and recall metrics. For the Gas dataset, we note differences of 2.26%, 2.49%, 2.00%, and 2.66% across the accuracy, F1 score, precision, and recall metrics. For the CICIDS dataset, our approach and the CPU baseline do not differ in accuracy, F1 score, precision, or recall. This is because all the features used to train the scikit-learn decision tree are integers, so no quantized representation of features is needed as with the previous three datasets. Our approach achieves identical effectiveness scores as the P4 baseline on the Mine and CICIDS datasets.

5.2 Latency

From the latency data we collected, we provide zoomed-in empirical cumulative distribution functions (eCDF) for each dataset, network link utilization, and CPU load in Figure 4. We also provide more concrete numbers on the 50th, 99th, and 99.9th percentiles across each dataset, link utilization, and CPU load in Tables 4 and 5. From our experiments, we make the following observations. We generally see a significant gap in the latency measurements between ML-NIC and the CPU baseline and a very small gap between ML-NIC and Xavier et al. (2021)'s approach. Specifically, we found that ML-NIC's latency can be at least 132.62 μs faster than the CPU baseline and 1.35 μs faster than Xavier et al. (2021)'s approach in the 50th percentile. However, there is a significant difference in the tails between ML-NIC and Xavier et al. (2021)'s approach, suggesting that ML-NIC has a stronger latency guarantee. Based on the 99.9th percentiles, we see that the tail latency of Xavier et al. (2021)'s approach can be at least $1.53\times$ larger than ML-NIC's tail.

Looking at impact of high network link utilization and CPU load, we observe very minimal fluctuation in the eCDFs of the ML-NIC and Xavier et al. (2021)'s approach. This suggests that both approaches are robust against high network link utilization and CPU load on these datasets. But, there is a more noticeable impact of high network link utilization and CPU load on the CPU baseline. As the network link utilization increases, we tend to see more probability mass shift towards the higher latency in the eCDF. With respect to the 99.9th latency percentile, we see an increase of at least $1.82\times$ from 0% link utilization to 99% link utilization. Concerning the increases in CPU load, there is a more significant shift in the eCDF curves

[2] Bold values indicate best values found for a given metric during experiments

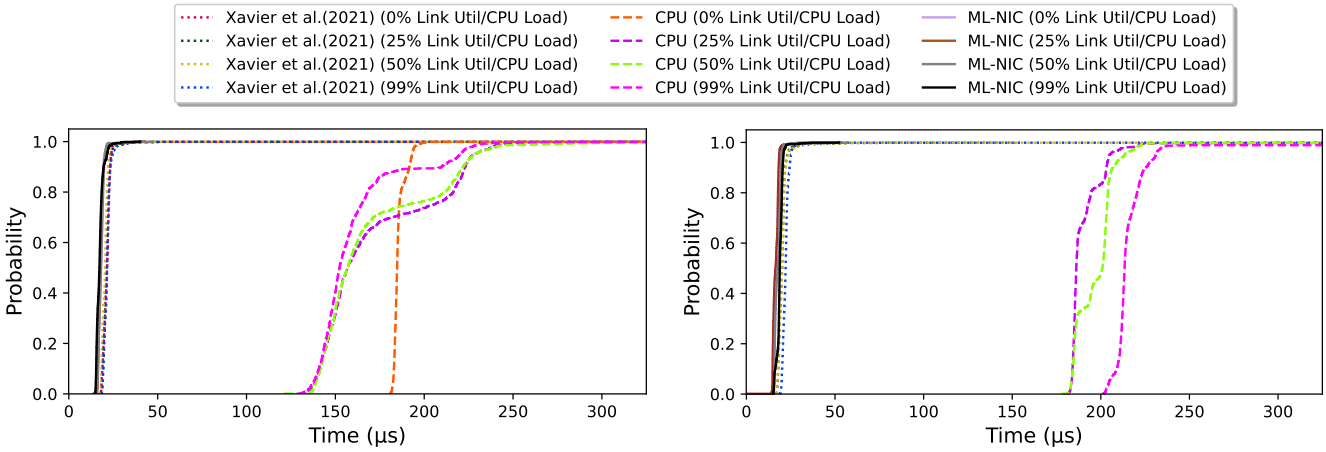

Figure 4a. Mine Tcpreplay

Figure 4b. Mine Stress-Ng

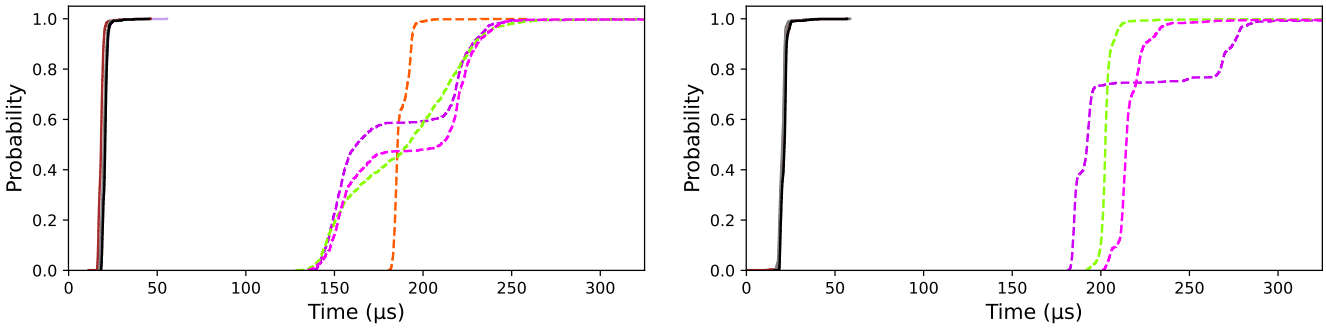

Figure 4c. Landsat Tcpreplay

Figure 4d. Landsat Stress-Ng

towards higher latencies. Referring to the 99.9th percentile, there is a latency increase of at least 20.27× from 0% CPU load to 99% CPU load.

Table 4. Latency Measurements Using Tcpreplay (μs)

Dataset	Link Util (%)	Percentile	CPU	Xavier et al. (2021)	ML-NIC
Mine	0	50	184.77	21.55	**18.00**
		99	197.39	25.41	**23.60**
		99.9	202.94	71.26	**35.54**
	25	50	156.93	21.30	**17.73**
		99	243.95	27.35	**23.63**
		99.9	285.89	65.31	**36.11**
	50	50	156.96	20.41	**18.15**
		99	267.83	27.33	**21.53**
		99.9	364.59	60.37	**31.05**
	99	50	152.32	21.55	**17.24**
		99	230.96	31.10	**24.45**
		99.9	454.20	70.65	**36.48**
Landsat	0	50	185.61	N/A	**19.99**
		99	200.27	N/A	**24.88**

		99.9	260.64	N/A	**39.63**
	25	50	162.39	N/A	**18.27**
		99	247.78	N/A	**22.70**
		99.9	365.45	N/A	**36.65**
	50	50	191.58	N/A	**19.82**
		99	258.51	N/A	**23.87**
		99.9	458.64	N/A	**42.08**
	99	50	209.91	N/A	**20.63**
		99	247.88	N/A	**24.96**
		99.9	503.92	N/A	**37.40**
Gas	0	50	185.99	N/A	**18.91**
		99	198.86	N/A	**26.57**
		99.9	287.77	N/A	**48.31**
	25	50	153.22	N/A	**20.60**
		99	255.51	N/A	**27.00**
		99.9	367.69	N/A	**41.65**
	50	50	159.18	N/A	**20.13**
		99	251.86	N/A	**28.91**
		99.9	410.64	N/A	**45.20**
	99	50	161.33	N/A	**22.11**
		99	244.74	N/A	**25.35**
		99.9	523.63	N/A	**43.98**
CICIDS	0	50	179.30	19.58	**17.21**
		99	189.71	23.86	**21.66**
		99.9	198.83	54.91	**31.70**
	25	50	154.72	20.00	**15.97**
		99	224.22	26.66	**19.76**
		99.9	277.53	55.06	**29.29**
	50	50	155.85	18.94	**17.55**
		99	225.52	23.11	**20.68**
		99.9	442.80	55.48	**33.41**
	99	50	154.81	21.91	**15.99**
		99	226.99	28.21	**20.39**
		99.9	483.02	51.85	**33.80**

Table 5. Latency Measurements Using Stress-Ng (μs)

Dataset	CPU Load (%)	Percentile	CPU	Xavier et al. (2021)	ML-NIC
Mine	25	50	186.16	20.16	**16.45**
		99	221.97	31.82	**20.43**
		99.9	343.53	74.56	**32.35**
	50	50	200.85	20.12	**17.69**
		99	222.64	31.16	**20.96**
		99.9	535.06	67.87	**34.50**
	99	50	213.40	22.12	**18.84**

		99	332.46	27.96	**22.66**
		99.9	7979.24	76.50	**36.86**
Landsat	25	50	192.30	N/A	**20.69**
		99	295.65	N/A	**24.48**
		99.9	405.26	N/A	**41.19**
	50	50	202.55	N/A	**20.23**
		99	215.20	N/A	**24.12**
		99.9	7582.86	N/A	**38.09**
	99	50	214.34	N/A	**21.54**
		99	260.63	N/A	**25.80**
		99.9	9008.55	N/A	**40.27**
Gas	25	50	183.90	N/A	**19.17**
		99	201.89	N/A	**27.42**
		99.9	469.44	N/A	**45.69**
	50	50	203.31	N/A	**23.01**
		99	238.13	N/A	**26.94**
		99.9	8852.89	N/A	**46.95**
	99	50	218.32	N/A	**19.40**
		99	251.67	N/A	**25.54**
		99.9	16182.94	N/A	**46.74**
CICIDS	25	50	155.58	20.13	**18.17**
		99	201.23	23.99	**21.31**
		99.9	357.74	51.48	**31.69**
	50	50	167.66	19.88	**18.18**
		99	184.74	24.39	**22.68**
		99.9	481.76	50.01	**30.33**
	99	50	175.56	20.15	**17.16**
		99	198.38	24.00	**21.83**
		99.9	4030.55	59.39	**32.74**

5.3 Throughput

As seen in Figure 5, there is a significant improvement in throughput with our approach compared to the CPU baseline. In our approach, we note $24.80\times$, $19.30\times$, $16.95\times$, and $20.11\times$ more packets per minute compared to the CPU baseline across the Mine, Landsat, Gas, and CICIDS datasets. Furthermore, our approach yields moderately higher throughput than the P4 baseline in the Mine and CICIDS dataset. In our approach, we observe $1.26\times$ and $1.11\times$ more packets per minute compared to the P4 baseline for the Mine and CICIDS datasets.

5.4 Hardware Utilization

From our hardware utilization experiment, we report the minimum, maximum, and average CPU and server host RAM utilization in Table 6. We also report the SmartNIC memory utilization as a constant, since dynamic memory allocation is not available on the AgilioCX 2×25 GbE SmartNIC. Since we are not able to directly measure the SmartNIC RAM used for the CPU baselines, we approximate it. Our approximation takes an unweighted average of the ratio of size of SmartNIC firmware for the CPU baselines over the size

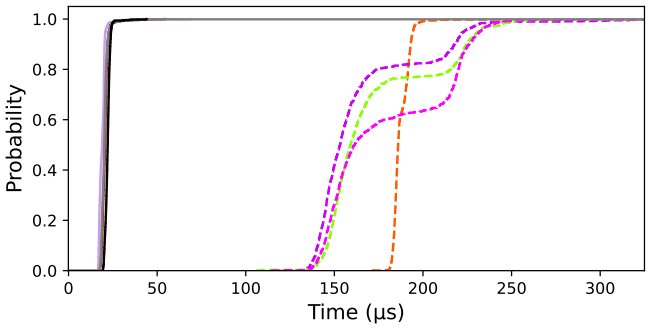

Figure 4e. Gas Tcpreplay

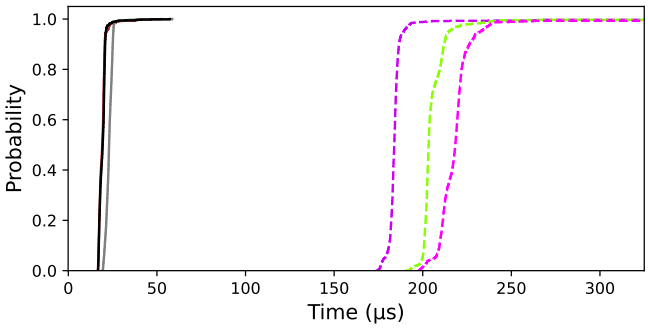

Figure 4f. Gas Stress-Ng

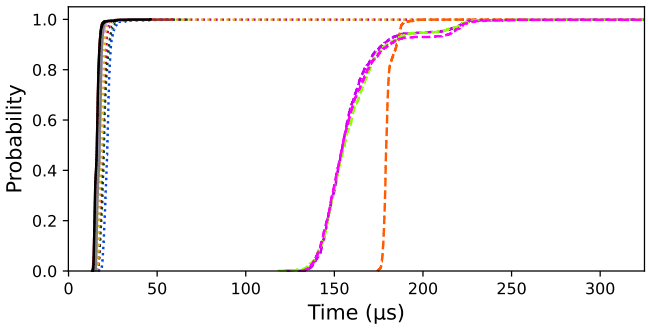

Figure 4g. CICIDS Tcpreplay

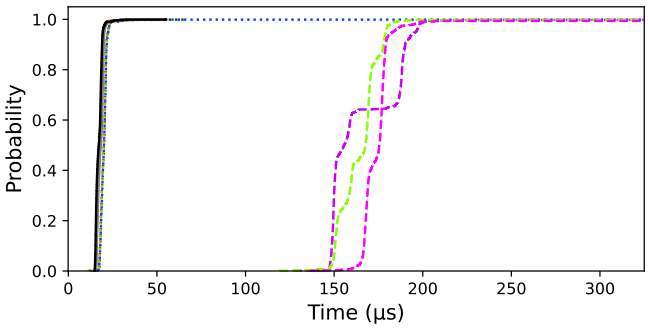

Figure 4h. CICIDS Stress-Ng

Figure 4. Figures 4a, 4c, 4e, and 4g depict the eCDFs for the latency experiments conducted using Tcpreplay to saturate the network link on all the datasets. Figures 4a, 4c, 4e, and 4g depict the eCDFs for the latency experiments conducted using Stress-Ng to generate a CPU load on all the datasets.

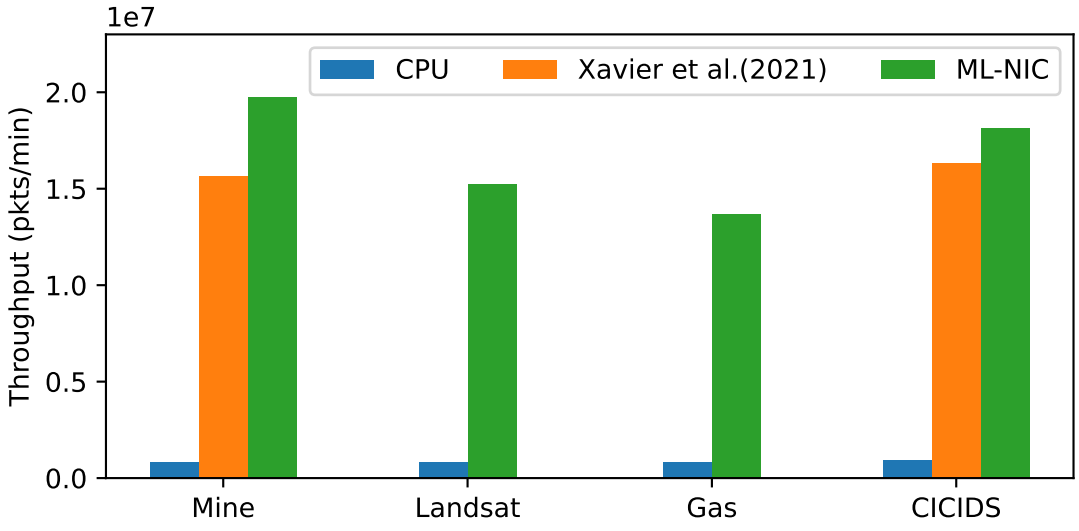

Figure 5. This figure provides a bar graph of the throughputs observed with our approach and two baselines.

of the SmartNIC firmware for the P4 baselines and our approach multiplied by the SmartNIC RAM used by those models for each of the datasets.

Table 6. Hardware Utilization Measurements

Dataset	Measure	CPU	Xavier et al. (2021)	ML-NIC
Mine	Min CPU (%)	**0.25**	**0.25**	**0.25**
	Avg CPU (%)	3.82	0.77	**0.52**
	Max CPU (%)	7.56	7.75	**0.94**
	Min Host RAM (MB)	2226.76	2147.76	**1908.97**
	Avg Host RAM (MB)	2228.43	2150.75	**1910.71**
	Max Host RAM (MB)	2230.56	2153.44	**1911.81**
	SmartNIC RAM (MB)	96.15	973.00	**21.13**
Landsat	Min CPU (%)	**0.25**	N/A	**0.25**
	Avg CPU (%)	3.81	N/A	**0.51**
	Max CPU (%)	7.53	N/A	**0.88**
	Min Host RAM (MB)	2226.50	N/A	**1909.86**
	Avg Host RAM (MB)	2229.00	N/A	**1910.69**
	Max Host RAM (MB)	2231.77	N/A	**1911.70**
	SmartNIC RAM (MB)	96.15	N/A	**21.12**
Gas	Min CPU (%)	0.25	N/A	**0.19**
	Avg CPU (%)	3.78	N/A	**0.51**
	Max CPU (%)	7.46	N/A	**0.88**
	Min Host RAM (MB)	2223.59	N/A	**1908.86**
	Avg Host RAM (MB)	2226.08	N/A	**1909.83**
	Max Host RAM (MB)	2228.07	N/A	**1910.98**
	SmartNIC RAM (MB)	96.15	N/A	**21.122**
CICIDS	Min CPU (%)	**0.19**	0.25	0.25
	Avg CPU (%)	3.68	0.80	**0.52**
	Max CPU (%)	7.20	9.09	**0.81**
	Min Host RAM (MB)	2225.02	2077.43	**1907.22**
	Avg Host RAM (MB)	2227.38	2078.88	**1908.36**
	Max Host RAM (MB)	2230.01	2080.04	**1909.21**
	SmartNIC RAM (MB)	96.15	973.00	**21.12**

From Table 6, we see that ML-NIC consistently uses lower resources for average and maximum host system's CPU, host system's RAM, and SmartNIC's RAM usage compared to the CPU baseline and Xavier et al. (2021)'s method. In the case where there the CPU baseline has slightly lower minimum CPU usage than the ML-NIC on the CICIDS dataset, this measurement likely corresponds to some degree of randomness in the measurement, since ML-NIC also achieved the same minimum CPU usage on the Gas dataset. In addition to the CPU baseline having a higher maximum CPU usage than ML-NIC by at least $7.91\times$, we also observe that Xavier et al. (2021)'s method can achieve similar or higher levels of maximum CPU usage. We attribute this to the runtime environment (RTE) server that is running on our host system, which is needed to run P4 code. We believe this also accounts for slightly higher host system RAM usage. For the SmartNIC's RAM usage, we observe our proxy for the CPU baseline to be lower than Xavier et al. (2021)'s method. Since the firmware running on the SmartNIC for the CPU baseline runs as a regular NIC, the SmartNIC would not require additional memory beyond storing extracting packet headers into local memory or general purpose registers. Furthermore, the larger SmartNIC RAM usage from Xavier et al.

(2021)'s method likely occurs because their approach involves running packet collect, feature computation, and result aggregation on every microengine. Since ML-NIC distributes these operations across multiple microengines, the resulting SmartNIC RAM usage would be lower by at least $46.05\times$.

6 DISCUSSION

6.1 Generalization

The work focuses on converting trained scikit-learn decision trees into Micro-C for deployment onto a SmartNIC. We focused on the Netronome AgilioCX 2×25 GbE. Deployment across different SmartNICs (assuming Micro-C support) may require significant code changes to accommodate the resources available on the card compared to the baselines.

6.2 Benefits

Despite their resource constraints compared to a host system's CPU, SmartNICs show potential as alternative hardware for deploying appropriately sized decision tree models. Deploying the decision tree model onto the SmartNIC, which brings it closer to the network edge, saves latency time by removing the need to transfer data over the PCIe bus from the NIC to the CPU without additional hardware. Furthermore, the lower-level programming used in our approach compared to the P4 baseline allows us to leverage device parallelism to deploy larger decision trees.

6.3 Scope

Our work only considers deploying decision tree models trained for various tasks onto a SmartNIC. Improvements in any specific use case are beyond the scope of our work.

6.4 Limitations

Our method's limitations depend on the SmartNIC's memory, computational, asynchronous I/O, and data rate constraints. Within the Netronome AgilioCX 2×25 GbE, the primary limits are the number of microengines (60), data rate (25 GbE), and number of hardware signals (15 per thread). While the constraint on microengines can be mitigated by assigning multiple features to a microengine, the memory (i.e., number of transfer registers) and asynchronous I/O (i.e., number of hardware signals) constraints limit the depth of the trained decision tree to 480 leaf nodes. Our current implementation is limited to decision trees with 256 leaf nodes, since more complex firmware is required to assign a hardware signal to a greater number of transfer registers.

6.5 Offloading More Models

6.5.1 Decision Tree

In addition to the decision tree models we have deployed in this work, we provide some more insights on other decision trees that our current work can offload onto a SmartNIC via an ablation study. In our ablation study, we use the CICIDS datasets to construct 10 decision trees with a constraint on the maximum number of leaf nodes between 16 and 256, based on the leaf node limit we mentioned in Section 3. For each decision tree, we look at the depth, number of nodes, number of leaf nodes, size of the pickle file, size of the firmware file, and SmartNIC RAM usage. We present our findings in Figure 6 below. Note that we scaled some of the measurements by a factor of 10, so the trends in the some of the decision tree parameters would be more clear.

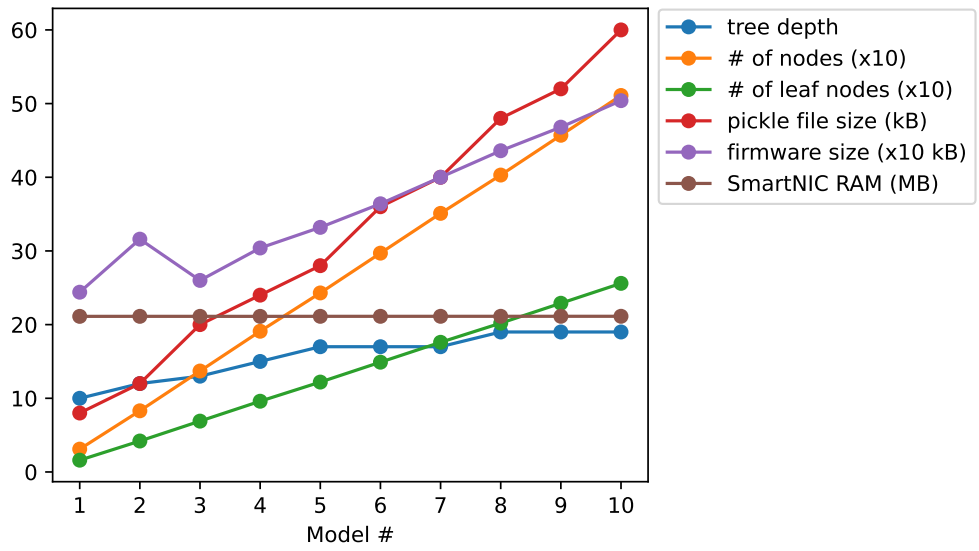

Figure 6. This figures shows how ML-NIC scales with respect to different decision tree parameters.

From Figure 6, we observe the following. First, we note the slow inclination, followed by a brief declination, then continued inclination in the usage of SmartNIC RAM. We also observe a similar pattern of inclination, declination, then inclination again in the trend for the model firmware size. We attribute this to how we compiled two instances of the model 1 and 2 per island to maximize our usage of the computation resources on the SmartNIC. For the remaining models, we only compiled one instance per island. Since models 1 and 2 have two instances compiled per island, more RAM and instruction memory would be needed to store the labels for the leaf nodes and decision tree logic. Based on the rate of inclination between firmware size and SmartNIC RAM usage, we see that a primary concern for offloading larger models is the amount of instruction memory available per microengine. After model 2, we see that the (scaled) trend for firmware size grows slower than that of the size of model pickle file and number of nodes by 1.64 and 1.53 and grows faster than the trend for decision tree depth by 4.07 and number of leaf nodes by 1.30.

6.5.2 Other Machine Learning Models

Besides decision trees, we also consider approaches for executing inference with other machine learning models. Since inference for many popular machine learning models relies heavily on the matrix-vector multiplication operation, we look into techniques for efficient and effective matrix-vector multiplication that can be performed by a SmartNIC. In addition to conducting inference on models such as neural network and support vector machines for supervised tasks, we find an implementing a suitable matrix-vector multiplication method necessary for unsupervised learning, such as with implementing k-means using cosine similarity as the similarity measure instead of Euclidean distance. First, a naive approach that we consider is creating a lookup table per weight to match a feature value with the multiplication of that feature with the specific weight. In this approach, the feature computation microengines would be responsible for doing the multiplication lookup based on the weight and feature value, and the result aggregation microengines would sum up the multiplied weight-feature values to obtain the final result. However, this approach may not be feasible for problems that require a large number of weights due to memory constraints on the SmartNIC.

An alternative approach would be to consider using natural logarithm and the exponent function (i.e., e^x). Instead of storing a lookup table per feature, two lookup tables can be stored to approximately compute

the natural logarithm and exponent of the feature values based on their fixed range (i.e., we assume each feature is in the range $[0, 1]$ in Section 3). Then, each feature computation microengine would be operate on a specific feature by conducting a lookup for the natural logarithm of the feature, taking the sum of the natural logarithms of the weights and the feature value, and conduct a lookup of the exponent of the sum of the natural logarithm values. While this approach may resolve issues with the memory constraint, a large number of features requires multiple lookups to the memory region holding the tables (i.e., CLS or IMEM) that can congest the CPP bus. So, to avoid this issue, the lookup tables could be replaced with first-order taylor approximations of the natural logarithm and exponent functions for a specific number of reference points in the range $[0, 1]$, where the taylor approximations can be represented using additions and bit shifts. At the same time though, the use of first-order taylor approximations can result in more erroneous model predictions.

More recently though, work by Blalock and Guttag (2021) proposed a novel technique for matrix-matrix multiplication that used locality-sensitive hashing to determine suitable functions (denote as $g(A)$) that can be executed efficiently using balanced binary regression trees. Based on their findings and our current implementation for decision tree inference, we believe their approach to be a more promising direction for executing matrix-vector multiplication on a SmartNIC.

6.6 Productionization and Scaling

When deploying decision tree models in the real world, we consider factors such as model updates and scaling. Concerning model updates (i.e., models retrained on larger datasets), we still limit the number of leaf nodes to 256, which may or may not be helpful as a regularization technique to prevent decision tree overfitting. In order to deploy a new decision tree, the original decision tree (i.e., the model firmware file) needs to be unloaded from the SmartNIC, and then the firmware for the new decision tree can be loaded onto the SmartNIC. This means that SmartNIC would be inactive while unloading the old decision tree and loading the new decision tree. So, inference requests can not be handled by a SmartNIC during that time.

For scaling, we primarily focus on the first model deployment scenario (model fits on an island). We do not believe much scaling of SmartNIC resources can be done as the entire card is required for one instantiation of the model. In the first deployment scenario, though, based on the amount of network traffic, firmware can be developed to set aside some islands for the decision tree model and others for other tasks. However, like the model update case, old firmware would need to be unloaded and new firmware loaded. So, inference requests can not be handled by a SmartNIC while unloading old and loading new firmware.

6.7 Hardware Improvements

Considering SmartNICs with Micro-C support that possess additional hardware capabilities, we first note SmartNICs with additional programmable flow-processing microengines. A SmartNIC can include more programmable microengines in two ways: additional islands or microengines per island. With additional microengines per island, we expect a further reduction in latency for all three model deployment scenarios. This would happen because more models would be able to fit on an island, which would remove the need for communication between the microengines on different islands. Communication between microengines on different islands is more expensive than between microengines on the same island. With additional islands, we expect a further increase in throughput for model deployment scenario one (the model can fit on one island). More islands mean more instances of the model that can be instantiated on the SmartNIC, which would allow it to meet more inference requests. In either scenario, we expect larger trees (with

respect to the number of features) to be more easily deployed, given that each microengine would be responsible for analyzing fewer features.

Next, we consider a SmartNIC with additional transfer registers per microengine. More transfer registers means that fewer microengines would be needed to perform result aggregation based on the feature analysis conducted by the feature computation microengines. With greater availability of microengines, we could likely deploy larger trees on the SmartNIC.

Lastly, another hardware improvement we consider is the addition of one or more FPUs. Adding FPUs resolves the issue with minor effectiveness degradation that we observe with the current iteration of our work while likely maintaining similar latency and throughput performance that we've observed in this work.

7 RELATED WORK

Recently, several works have leveraged programmable data planes to make aspects of machine learning more efficient. These works can be split into two categories: model training and model inference. We focus on the latter. Within model inference, research efforts are focused on leveraging how entire or portions of the machine learning model inference process can be offloaded onto programmable data planes while maintaining adequate model performance. These implementations are most commonly conducted on programmable switches and SmartNICs.

7.1 Programmable Switch

For works that map machine learning models onto programmable switches, we generally observe a focus on specific models used to address certain tasks. We first note Net2Net (Siracusano and Bifulco, 2018) that proposed quantizing neural networks into binary neural networks, since they require operations that are readily available on modern switching chips. Rather than quantizing a trained neural network into a binary neural network, Qin et al. (2020) directly trained binary neural networks and mapped them onto the data planes of programmable switches using P4 to handle the network intrusion detection use case. As an alternative to binary neural network quantization, Dao et al. (2021) used neuron pruning to map neural networks onto programmable switches for the network intrusion use case. While the above neural network works solely considered a single programmable switch for deployment, Saquetti et al. (2021) implemented a neural network neuron distribution method across multiple programmable switches and coordinated inference of the model between the switches to optimize resource usage. Similarly, JointNIDS (Dao and Lee, 2022) also employed a distributed neural network inference approach. But, the neural network intrusion detection models were split into two sequential submodels with overlapping hidden units and mapped the submodels onto two programmable switches. The authors assigned one programmable switch to detect major network attacks, while the second handled the more subtle aspects of network traffic classification. Rather than staying within the constraints of off-the-shelf programmable network devices, Taurus (Swamy et al., 2022) extended the PISA architecture of programmable switches by adding custom hardware to support parallelism and additional operations (i.e., multiplication, nonlinear operations) needed to run neural network inference without any quantization.

Regarding tree-based models, pForest (Busse-Grawitz et al., 2019) developed a optimization technique to map a random forest classifier to a programmable switch in P4 for network flow classification. Furthermore, this approach adaptively switches out the current classifier with others based on the network flows observed. In addition to mapping a random forest classifier, Planter (Zheng and Zilberman, 2021) mapped a xgboost

and isolation forest classifier to programmable switches using overlapping trees to overcome some of the inefficiencies observed in pForest (Busse-Grawitz et al., 2019). Similar to pForest (Busse-Grawitz et al., 2019), SMASH (Kamath and Sivalingam, 2021) also focused on the network flow classification task and used an improved hash-and-store algorithm with a decision tree model for early flow classification. Also working with decision trees, pHeavy (Zhang et al., 2021) implemented trained decision trees on the data plane to reduce the overhead involved with communicating to the control plane in Software-Defined Networking (SDN) when classifying highly-congested network flows. In contrast to the other tree-based model offloading approaches that focus on network-related use cases, NetPixel (Siddique et al., 2021) implemented decision trees on P4 programmable switches to handle image classification. To address some of the issues with deployment of decision trees and other machine learning algorithms onto programmable data planes, Mousika (Xie et al., 2022) introduced a teacher-student knowledge distillation approach to translate machine learning models to binary decision trees, which are more suitable for mapping onto the data plane.

On top of supervised machine learning, the deployment of unsupervised learning algorithms onto programmable switches has also been explored. Clustreams (Friedman et al., 2021) used a combination of the quadtree data structure and a match+action table stored in Ternary Content Addressable Memory (TCAM) to cluster network traffic efficiently. In addition, ACC-Turbo (Alcoz et al., 2022) redesigned the original Aggregate-based Congestion Control (ACC) approach using online clustering and a scheduling algorithm to mitigate pulse-wave DDoS attacks.

Unlike the works above that focus on a specific machine learning algorithm type (i.e., neural networks, tree-based models, clustering, etc.), IIsy (Xiong and Zilberman, 2019) introduced mapping schemes for several machine learning algorithms, such as decision trees, k-means, naive bayes, and support vector machines, to the data plane using the match-action pipeline in programmable switches. Also, Hong et al. (2024) developed a feature engineering and model deployment strategy for tree-based models (i.e., decision trees, random forests, xgboost), k-nearest neighbor, and k-means to handle the high-frequency stock market trading task.

7.2 SmartNIC

Similar to the works that address deployment of machine learning model inference onto programmable switches, we see works about model inference onto SmartNICs that also consider neural networks and decision trees used for particular applications. Using the approach proposed by Net2Net (Siracusano and Bifulco, 2018), BaNaNa split (Sanvito et al., 2018) accelerated the inference of neural networks by splitting a neural network at its fully-connected layers, sending all prior layers to the host system's CPU for inference, and quantizing the fully-connected layers to run portion of the inference on the host system's SmartNIC. Different from the other works that primarily look into P4 implementations, N3IC (Siracusano et al., 2022) used Micro-C and P4 to map binary neural networks onto a greater variety of targets (i.e., SmartNICs) for traffic analysis use cases. Regarding tree-based models, Xavier et al. (2021) presented a framework for deploying decision tree models onto SmartNICs in P4. The authors demonstrated that their framework can achieve high accuracy (above 95%) in a network intrusion detection use case. While similar to our work, we note that ML-NIC works on a greater variety of use cases outside of network traffic analysis. Furthermore, ML-NIC's Micro-C implementations can parallelize the model inference process, which is not possible with P4. While the works on machine learning inference offloading for SmartNICs do not cover unsupervised learning to our knowledge, they do address traditional reinforcement learning.

Opal (Simpson and Pezaros, 2022) implemented online reinforcement learning onto a SmartNIC data plane, relying on classical reinforcement algorithms such as Sarsa (Sutton, 2018) and avoiding neural networks.

8 FUTURE DIRECTIONS

8.1 Implementing Additional Models

As mentioned in Section 6, our next step is to expand our framework to other machine learning algorithms, such as support vector machines and neural networks. We find that implementing the approximate matrix-matrix multiplication approach developed by Blalock and Guttag (2021) to be means of achieving this goal. In addition to matrix-matrix multiplication, there are floating point operations that are often performed for various models such as neural networks. Thus, in order to implement such models, future work may involve, either adding hardware support for these operations or using quantized operations as a default. We discuss the potential future work in this area in Section 8.2.

8.2 Improving the Floating-Point Representation

While our current approach leverages fixed-point 16-bit features and produces comparable model effectiveness scores to the baselines, it is possible that the proposed float-representation scheme can be improved without adding more hardware. One potential future direction to be explored is to look into the posit representation Gustafson and Yonemoto (2017) as a potential alternative, since it resolves the issue of NaN quantities observed in the standard floating-point representation. Also, while implementing the standard floating-point representation on the SmartNIC may seem like a viable solutions, we believe that the float-pointing representation standard would introduce additional latency due to the additional computation spent managing mantissa bits, exponent bits, and NaN quantities. In addition, adding any additional hardware support may cause added cost and energy consumption of SmartNIC, which defeats the purpose of using the SmartNIC in the first place. Thus, a software / algorithmic based approaches for performing floating point operations will be a great direction for future work.

8.3 Automating the Model Deployment

Furthermore, despite automating the process of decision tree logic extraction, the process of building the model mapping still mostly requires the developer to manually allocate cores as one of packet collection, feature computation, or result aggregation. We think the model mapping component can be made more efficient with additional code that considers the computation constraints of the SmartNIC and presents a mapping scheme to remove some of the tedious work in deploying a model onto the SmartNIC. Thus, a direction for future work may involve building a more sophisticated system that can perform model compilation, optimization and deployment automatically to the SmartNIC. This work can be further strengthened by adding a notion of distributed deployment and model inference across multiple SmartNICs located on multiple server.

8.4 Utilizing Different Types of SmartNICs

While this work primarily utilizes ASIC-based SmartNIC, it is possible to implement similar work on other ASIC-based and other types of SmartNICs, such as FPGA-based SmartNICs. While the optimizations we performed in this paper is specific to Netronome SmartNIC, the overall idea of mapping memory into different SmartNIC region is quite generic. Thus, a potential valuable future work is to perform similar

optimization strategies across different types of hardware implementations to understand the similarities and differences in the effectiveness of the proposed optimization and compilation strategies.

9 CONCLUSION

Low-latency model inference is a necessity for many time-sensitive machine learning applications. This paper demonstrates that ML-NIC is a suitable framework for performing machine learning model inference. Our evaluation of the first iteration of ML-NIC shows that it can deploy larger models than the state-of-the-art SmartNIC approach, can produce predictions at faster speeds with a minor loss in model effectiveness compared to the CPU solution, and is robust to high network utilization and CPU loads.

CONFLICT OF INTEREST STATEMENT

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

AUTHOR CONTRIBUTIONS

SC conceived the presented idea and provided computing resources. DCA and SC supervised the project, designed experiments, and validated the findings. RK conducted experiments, collected and analyzed data, and wrote the initial draft of the manuscript. All authors were involved in reviewing and revising the manuscript.

DATA AVAILABILITY STATEMENT

The datasets analyzed for this study can be found in the University of California, Irvine Machine Learning Repository [https://archive.ics.uci.edu/] and University of New Brunswick Canadian Institute for Cybersecurity [https://www.unb.ca/cic/].

REFERENCES

Alcoz, A. G., Strohmeier, M., Lenders, V., and Vanbever, L. (2022). Aggregate-based congestion control for pulse-wave ddos defense. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 693–706

Blalock, D. and Guttag, J. (2021). Multiplying matrices without multiplying. In *International Conference on Machine Learning* (PMLR), 992–1004

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

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., et al. (2014). Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture* (IEEE), 609–622

Choquette, J., Giroux, O., and Foley, D. (2018). Volta: Performance and programmability. *Ieee Micro* 38, 42–52

Corigine (2020). *Corigine NFP-4000 Flow Processor*. Tech. rep., Corigine

Dang, H. T., Bressana, P., Wang, H., Lee, K. S., Zilberman, N., Weatherspoon, H., et al. (2020). P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking* 28, 1726–1738

Dao, T.-N., Hoang, V.-P., Ta, C. H., et al. (2021). Development of lightweight and accurate intrusion detection on programmable data plane. In *2021 International Conference on Advanced Technologies for Communications (ATC)* (IEEE), 99–103

Dao, T.-N. and Lee, H. (2022). Jointnids: Efficient joint traffic management for on-device network intrusion detection. *IEEE Transactions on Vehicular Technology* 71, 13254–13265

Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., et al. (2018). A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (IEEE), 1–14

Friedman, R., Goaz, O., and Rottenstreich, O. (2021). Clustreams: Data plane clustering. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 101–107

Gustafson, J. L. and Yonemoto, I. T. (2017). Beating floating point at its own game: Posit arithmetic. *Supercomputing frontiers and innovations* 4, 71–86

He, Z., Sidler, D., István, Z., and Alonso, G. (2018). A flexible k-means operator for hybrid databases. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)* (IEEE), 368–3683

Hong, X., Zheng, C., Zohren, S., and Zilberman, N. (2024). Accelerating machine learning for trading using programmable switches (IOS Press), Frontiers in Artificial Intelligence and Applications, 3429–3436

Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., et al. (2017). Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12

Kamath, R. and Sivalingam, K. M. (2021). Machine learning based flow classification in dcns using p4 switches. In *2021 International Conference on Computer Communications and Networks (ICCCN)* (IEEE), 1–10

Kim, C., Sivaraman, A., Katta, N., Bas, A., Dixit, A., Wobker, L. J., et al. (2015). In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*. vol. 15, 1–2

Langlet, J. (2019). Towards machine learning inference in the data plane

Leo Breiman, C. J. S., Jerome Friedman and Olshen, R. (1984). *Classification and Regression Trees* (Chapman and Hall/CRC)

Liu, Y. H. (2017). *Python machine learning by example* (Packt Publishing Ltd)

Netronome Systems, I. (2024). *Agilio CX 2x25GbE SmartNIC*. Tech. rep., Netronome Systems, Inc.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research* 12, 2825–2830

Qin, Q., Poularakis, K., Leung, K. K., and Tassiulas, L. (2020). Line-speed and scalable intrusion detection at the network edge via federated learning. In *2020 IFIP Networking Conference (Networking)* (IEEE), 352–360

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning* 1, 81–106

Rodríguez-Luján, I., Fonollosa, J., Vergara, A., Homer, M. L., and Huerta, R. (2014). On the calibration of sensor arrays for pattern recognition using the minimal number of experiments. *Chemometrics and Intelligent Laboratory Systems* 130, 123–134

[Dataset] Roth, D. (2016). Decision trees

Sanvito, D., Siracusano, G., and Bifulco, R. (2018). Can the network be the ai accelerator? In *Proceedings of the 2018 Morning Workshop on In-Network Computing*. 20–25

Saquetti, M., Canofre, R., Lorenzon, A. F., Rossi, F. D., Azambuja, J. R., Cordeiro, W., et al. (2021). Toward in-network intelligence: Running distributed artificial neural networks in the data plane. *IEEE Communications Letters* 25, 3551–3555

Sharafaldin, I., Lashkari, A. H., Ghorbani, A. A., et al. (2018). Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp* 1, 108–116

Siddique, H., Neves, M., Kuzniar, C., and Haque, I. (2021). Towards network-accelerated ml-based distributed computer vision systems. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)* (IEEE), 122–129

Simpson, K. A. and Pezaros, D. P. (2022). Revisiting the classics: Online rl in the programmable dataplane. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium* (IEEE), 1–10

Siracusano, G. and Bifulco, R. (2018). In-network neural networks. *arXiv preprint arXiv:1801.05731*

Siracusano, G., Galea, S., Sanvito, D., Malekzadeh, M., Antichi, G., Costa, P., et al. (2022). Re-architecting traffic analysis with neural network interface cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 513–533

[Dataset] Srinivasan, A. (1993). Statlog (Landsat Satellite). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C55887

Sutton, R. S. (2018). Reinforcement learning: an introduction. *A Bradford Book*

Swamy, T., Rucker, A., Shahbaz, M., Gaur, I., and Olukotun, K. (2022). Taurus: a data plane architecture for per-packet ml. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1099–1114

Tong, D., Qu, Y. R., and Prasanna, V. K. (2017). Accelerating decision tree based traffic classification on fpga and multicore platforms. *IEEE Transactions on Parallel and Distributed Systems* 28, 3046–3059

Wray, S. (2014). *The Joy of Micro-C*. Tech. rep., Netronome Systems, Inc.

Xavier, B. M., Guimarães, R. S., Comarela, G., and Martinello, M. (2021). Programmable switches for in-networking classification. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications* (IEEE), 1–10

Xie, G., Li, Q., Dong, Y., Duan, G., Jiang, Y., and Duan, J. (2022). Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications* (IEEE), 1938–1947

Xiong, Z. and Zilberman, N. (2019). Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*. 25–33

Yilmaz, C., Kahraman, H. T., and Söyler, S. (2018). Passive mine detection and classification method based on hybrid model. *IEEE Access* 6, 47870–47888. doi:10.1109/ACCESS.2018.2866538

Zhang, B., Kannan, R., Prasanna, V., and Busart, C. (2023). Accelerating gnn-based sar automatic target recognition on hbm-enabled fpga. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)* (IEEE), 1–7

Zhang, X., Cui, L., Tso, F. P., and Jia, W. (2021). pheavy: Predicting heavy flows in the programmable data plane. *IEEE Transactions on Network and Service Management* 18, 4353–4364

Zheng, C. and Zilberman, N. (2021). Planter: seeding trees within switches. In *Proceedings of the SIGCOMM '21 Poster and Demo Sessions* (New York, NY, USA: Association for Computing Machinery), SIGCOMM '21. 12–14. doi:10.1145/3472716.3472846

APPENDIX

A Packet Collection Implementation

An example implementation for packet collection is provided below. At the start of the main function, the microengine signals the CTM packet engine that it is ready to receive a packet. Once the MAC block has received and preprocessed a packet, it is transferred to the microengine's corresponding CTM buffer and an MU buffer (if needed). The remainder of the code then validates the packet as a model input packet, parses the features embedded in the packet payload, and transfers the features to various microengines. After transferring the packets, the result aggregation microengine waits to be signaled that the final classification is produced. Then, the original model input packet is modified with the classification result, some post-processing is performed, and the CTM packet engine is informed that the packet is ready to be sent out.

Listing 1. Example Packet Collection Code

```
#include <nfp.h>
#include <nfp/remote_me.h>
#include <nfp/mem_bulk.h>
... /* Some more packet imports */
#include <net/hdr_ext.h>
#include "config.h"

/*
Read packet data from memory in xfer registers as a two byte
offset so that the payload get aligned to a word boundary. This
makes the header extraction code more efficient.
*/
#define PKT_START_OFF 2


/* A structure used for extracting, the different protocol header  */
struct pkt_hdr {
    struct eth_hdr  eth;
    struct ip4_hdr  ip4;
};

// Signals for interacting with other MEs
SIGNAL local_signal;

__remote SIGNAL remote_signal1;
__remote __xread uint32_t remote_xfer1;

__remote SIGNAL remote_signal2;
__remote __xread uint32_t remote_xfer2;

... /* Repeat similar variable for each feature */

```

```
815 __visible SIGNAL complete_signal;
816 __visible __xread uint32_t complete_xfer;
817
818 __intrinsic void proc_pkt(__mem40 char *buf_addr, __gpr uint32_t
819     buf_off) {
820
821     // Extract the headers first before the payload
822     __xread uint32_t pkt_buf[17];
823     __xwrite uint32_t data_out[15];
824     __lmem uint32_t src_buf[13];  // just store the headers
825     __gpr struct pkt_hdr eh;
826     __gpr uint32_t csum_prepend;
827     __gpr int src_off = buf_off;
828     __gpr int res;
829     __gpr int len;
830
831     // Extract the packet
832     mem_read32(pkt_buf, buf_addr + buf_off - PKT_START_OFF, sizeof(
833         pkt_buf));
834
835     /* Copy xfer register to a Local Memory buffer for easier
836        extraction */
837     reg_cp(src_buf, pkt_buf, sizeof(src_buf));
838     src_off = PKT_START_OFF;
839
840     /*
841      * Handle the checksum prepend if configured
842      */
843 #ifdef CFG_RX_CSUM_PREPEND
844     /* read the MAC parsing info for CSUM (first 4B are timestamp) */
845     csum_prepend = pkt_csum_read(pkt_buf, PKT_START_OFF + 4);
846     src_off += MAC_PREPEND_BYTES;
847
848     if (NFP_MAC_RX_CSUM_L3_SUM_of(csum_prepend) ==
849         NFP_MAC_RX_CSUM_L3_IPV4_FAIL) {
850         /* L3 checksum is wrong */
851         return;
852     }
853
854     if ((NFP_MAC_RX_CSUM_L4_SUM_of(csum_prepend) ==
855          NFP_MAC_RX_CSUM_L4_TCP_FAIL) ||
856         (NFP_MAC_RX_CSUM_L4_SUM_of(csum_prepend) ==
857          NFP_MAC_RX_CSUM_L4_UDP_FAIL)){
858         /* L4 checksum is wrong */
859         return;
```

```
860     }
861 #endif
862
863     // Layer 2 Extraction
864     res = he_eth(src_buf, src_off, &eh.eth);
865     len = HE_RES_LEN_of(res);
866     src_off += len;
867
868     // Check if ethernet header is correct
869     if (((__gpr uint16_t *)&eh.eth.dst)[0] == 0x0015
870         && ((__gpr uint16_t *)&eh.eth.dst)[1] == 0x4d13
871         && ((__gpr uint16_t *)&eh.eth.dst)[2] == 0x79ac
872         && net_eth_is_uc_addr(&eh.eth.src)
873         && eh.eth.type == 0x4d49) {
874
875         // Edit Ethernet Header
876         ((__lmem struct eth_hdr *)((__lmem char *)src_buf+10))->dst =
877             eh.eth.src;
878         ((__lmem struct eth_hdr *)((__lmem char *)src_buf+10))->src =
879             eh.eth.dst;
880         ((__lmem struct eth_hdr *)((__lmem char *)src_buf+10))->type
881            = 0x4d4f;
882
883         // Edit IP Header
884         res = he_ip4(src_buf, src_off, &eh.ip4);
885         ((__lmem struct ip4_hdr *)(src_buf+6))->dst = eh.ip4.src;
886         ((__lmem struct ip4_hdr *)(src_buf+6))->src = eh.ip4.dst;
887
888         // Copy header and payload to data_out
889         reg_cp(data_out, src_buf+2, 44);
890         reg_cp(data_out+11, pkt_buf+13, 16);
891
892         // Inform ME 1 that data is ready
893         remote_me_reg_write_signal_remote(data_out+11, 32, 1, 0,
894                                           __xfer_reg_number(&
895                                              remote_xfer1, __nfp_meid
896                                              (32, 1)) + (__ctx() * 32)
897                                              ,
898                                           sizeof(remote_xfer1), &
899                                              local_signal);
900
901         // Inform ME 2 that data is ready
902         remote_me_reg_write_signal_remote(data_out+11, 32, 2, 0,
```

```
903                                                 __xfer_reg_number(&
904                                                    remote_xfer2, __nfp_meid
905                                                    (32, 2)) + (__ctx() * 32)
906                                                    ,
907                                                 sizeof(remote_xfer2), &
908                                                    local_signal);
909
910         ... /* Repeat microengine data transfer for each feature */
911
912         // Wait for ME 9 to finish
913         __wait_for_all(&complete_signal);
914         reg_cp(data_out+11, &complete_xfer, sizeof(complete_xfer));
915         mem_write64(data_out, buf_addr + buf_off + 6, 48);
916     }
917 }
918
919 int main(void) {
920     __mem40 char *pbuf;
921     __xread struct nbi_meta_catamaran nbi_meta;
922     __xread struct nbi_meta_pkt_info *pi = &nbi_meta.pkt_info;
923     __gpr int in_port, pkt_off;
924     __gpr struct pkt_ms_info msi;
925
926     __assign_relative_register(&local_signal, 1);
927     __assign_relative_register(&complete_signal, 10);
928     __implicit_write(&complete_signal, sizeof(complete_signal));
929     __implicit_write(&complete_xfer, sizeof(complete_xfer));
930
931     for(;;) {
932         /* Receive packet */
933         pkt_nbi_recv(&nbi_meta, sizeof(nbi_meta));
934         in_port = MAC_TO_PORT(nbi_meta.port);
935         pbuf = pkt_ctm_ptr40(pi->isl, pi->pnum, 0);
936
937         /* Collect features from pkt */
938         pkt_off = PKT_NBI_OFFSET;
939         proc_pkt(pbuf, pkt_off);
940
941         /* Send packet */
942         // Write the MAC egress CMD and adjust offset and len as
943            needed
944         pkt_off += MAC_PREPEND_BYTES;
945         pkt_mac_egress_cmd_write(pbuf, pkt_off, 1, 1);
946
947         pkt_off -= 4;
```

```
        msi = pkt_msd_write(pbuf, pkt_off);
        pkt_nbi_send(pi->isl, pi->pnum, &msi,
                     pi->len - MAC_PREPEND_BYTES + 4,
                     NBI, PORT_TO_TMQ(in_port),
                     nbi_meta.seqr, nbi_meta.seq, PKT_CTM_SIZE_256);
    }

    return 0;
}
```

B Feature Computation Implementation

The code listing below exemplifies how we program a microengine for feature computation. Once the packet collection microengine signals the microengine and receives a feature to analyze, it executes the logic learned by the trained decision for that specific feature as a series of if-else statements to determine which leaf nodes can be reached based on the value taken on by the feature in question. Once the possible leaf nodes are calculated, the feature computation microengine transmits the possible leaf nodes to the result aggregation microengine.

Listing 2. Example Feature Computation Code

```
#include <nfp.h>
#include <stdint.h>
#include <std/reg_utils.h>
#include <nfp/me.h>
#include <nfp/remote_me.h>
#include <nfp/cls.h>

__visible SIGNAL remote_signal2;
__visible __xread uint32_t remote_xfer2;

__remote SIGNAL result_signal2;
__remote __xread uint32_t result_xfer2[4];

SIGNAL local_signal;

int main(void) {
    __gpr uint32_t feature;
    __gpr uint32_t result_gpr[4];
    __xwrite uint32_t result_write[4];

    __assign_relative_register(&remote_signal2, 1);
    __assign_relative_register(&local_signal, 3);
    __implicit_write(&remote_signal2, sizeof(remote_signal2));
    __implicit_write(&remote_xfer2, sizeof(remote_xfer2));

    for (;;) {
```

```
990         reg_set(result_gpr, 0x3ffff, sizeof(uint32_t));
991         reg_set(result_gpr+1, 0xffffffff, sizeof(uint32_t));
992         reg_set(result_gpr+2, 0xffffffff, sizeof(uint32_t));
993         reg_set(result_gpr+3, 0xffffffff, sizeof(uint32_t));
994
995         __wait_for_all(&remote_signal2);
996         reg_cp(&feature, &remote_xfer2, sizeof(feature));
997         feature &= 0xffff;
998
999         // Decision Tree logic
1000        if (!(feature <= 5585))
1001            result_gpr[3] &= ~(1 << 3);
1002
1003        if (!(feature <= 5585))
1004            result_gpr[3] &= ~(1 << 4);
1005
1006        if (!(feature <= 5585))
1007            result_gpr[3] &= ~(1 << 5);
1008
1009        if (!(feature <= 5585))
1010            result_gpr[3] &= ~(1 << 6);
1011
1012        if (!(feature <= 5585 && feature <= 4841))
1013            result_gpr[3] &= ~(1 << 7);
1014
1015        if (!(feature <= 5585 && feature > 4841))
1016            result_gpr[3] &= ~(1 << 8);
1017
1018        if (!(feature <= 5585 && feature <= 3351 && feature <= 1117))
1019            result_gpr[3] &= ~(1 << 9);
1020
1021        ... /* Remaining decision tree feature logic */
1022
1023        // Transfer result to result aggregation microengine
1024        reg_cp(&result_write, &result_gpr, sizeof(result_write));
1025        remote_me_reg_write_signal_remote(&result_write, 32, 9, 0,
1026                                          __xfer_reg_number(&
1027                                            result_xfer2, __nfp_meid
1028                                            (32, 9)) + (__ctx() * 32)
1029                                            ,
1030                                          sizeof(result_xfer2), &
1031                                            local_signal);
1032    }
1033
1034    return 0;
```

}

C Result Aggregation Implementation

Below, we provide an example implementation for result aggregation. Once the microengine receives all possible leaf nodes based on the values taken on by each feature from the feature computation microengines, it takes the single possible leaf node predicted by all the feature computation microengines and searches for the corresponding class label. The result aggregation microengine transmits the label to the packet collection microengine to complete the inference process.

Listing 3. Example Result Aggregation Code

```
#include <nfp.h>
#include <stdint.h>
#include <std/reg_utils.h>
#include <nfp/me.h>
#include <nfp/remote_me.h>
#include <nfp/cls.h>

__visible SIGNAL result_signal1;
__visible __xread uint32_t result_xfer1[4];

... /* Repeat similar variables for each feature */

// For transfer back to packet collection microengine
__remote SIGNAL complete_signal;
__remote __xread uint32_t complete_xfer;

SIGNAL local_signal;

int main(void) {
    __gpr uint32_t results_gpr1[4];
    __gpr uint32_t results_gpr2[4];
    ... /* Repeat similar variables for each feature */
    __xwrite uint32_t final_result_write;

    __lmem uint8_t path_class[] = {1, 4, 4, 1, 4, 1, 1, 4, 5, 1, 5,
       1, 5, 5, 3, 3, 4, 1, 3, ...};

    __assign_relative_register(&result_signal1, 2);
    .../* Repeat similar function call for each feature */
    __assign_relative_register(&local_signal, 10);

    __implicit_write(&result_signal1, sizeof(result_signal1));
    __implicit_write(result_xfer1, sizeof(result_xfer1));
    .../* Repeat similar function calls for each feature */

```

```
    for (;;) {
        __wait_for_all(&result_signal1, &result_signal2, ...);

        // Collect all the possible paths
        reg_cp(results_gpr1, result_xfer1, sizeof(results_gpr1));
        ... /* Repeat similar function call for each feature */

        // Compute final path
        results_gpr8[0] = results_gpr1[0] & ...;
        ... /* Repeat similar computation for each 32-bit word */

        // Calculate leaf node index
        if (results_gpr8[0] != 0)
            results_gpr8[0] = ffs(results_gpr8[0]) + 96;
        else if (results_gpr8[1] != 0)
            results_gpr8[0] = ffs(results_gpr8[1]) + 64;
        ... /* Repeat similar computation for each 32-bit word */
        else
            results_gpr8[0] = ffs(results_gpr8[3]);

        // Get class prediction
        results_gpr8[0] = (uint32_t)(path_class[results_gpr8[0]]);
        reg_cp(&final_result_write, results_gpr8, sizeof(
           final_result_write));

        // Send final result to packet collection core
        remote_me_reg_write_signal_remote(&final_result_write, 32, 0,
            0,
                                          __xfer_reg_number(&
                                             complete_xfer, __nfp_meid
                                             (32, 0)) + (__ctx() * 32)
                                             ,
                                          sizeof(complete_xfer), &
                                             local_signal);
    }

    return 0;
}
```

D Model Deployment Implementation

Once all the Micro-C code files have been created, the following provides an example of deploying the mapped decision tree onto the SmartNIC. Each nfcc command processes a Micro-C code file and builds the corresponding .list file. This process is repeated for each microengine intended for use. Then the code is linked, where we explicitly specify which microengine will run which code file to get the final firmware file. Lastly, we load the firmware onto the SmartNIC and start running the firmware.

Listing 4. Micro-C Code Compilation onto SmartNIC example

```
# --- Building blm.list
nfas -t -W3 -R -lm 0 -C -chip nfp-4xxx-b0  -DBLM_CUSTOM_CONFIG -
    DNFP_LIB_ANY_NFAS_VERSION -I. -I/c_packetprocessing/microc/blocks -
    I/c_packetprocessing/microc/include -I/c_packetprocessing/microc/
    lib -I/components/standardlibrary/include -I/components/
    standardlibrary/microcode/include -I/components/standardlibrary/
    microcode/src -DBLM_CUSTOM_CONFIG -DSINGLE_NBI -DPKT_NBI_OFFSET= -
    DBLM_BLQ_EMEM_TYPE=emem -DNBII=8 -DBLM_INSTANCE_ID=0 -
    DBLM_INIT_EMU_RINGS -I. -I/dtree/mine/micro_c -I/c_packetprocessing
    /microc/blocks/blm/ -I/c_packetprocessing/microc/blocks/blm/_h -I/
    c_packetprocessing/microc/blocks/blm/_uc -o blm.list /
    c_packetprocessing/microc/blocks/blm/blm_main.uc

### Island 32 ####
# --- Building pkt_collect.list
nfcc -W3 -chip nfp-4xxx-b0 -Qspill=7 -Qnn_mode=1 -Qno_decl_volatile -
    single_dram_signal -Qnctx_mode=8 -FI config.h  -DBLM_CUSTOM_CONFIG
    -I. -I/dtree/mine/micro_c -I/c_packetprocessing/microc/include -I/
    c_packetprocessing/microc/lib -I/c_packetprocessing/microc/blocks/
    blm -I/c_packetprocessing/microc/blocks/blm/_h -I/components/
    standardlibrary/include -I/components/standardlibrary/microcode/
    include -Fepkt_collect_32.list /dtree/mine/micro_c/island32/
    pkt_collect_32.c /c_packetprocessing/microc/lib/nfp/libnfp.c /
    c_packetprocessing/microc/lib/std/libstd.c /c_packetprocessing/
    microc/lib/pkt/libpkt.c /c_packetprocessing/microc/lib/net/libnet.c
     /components/standardlibrary/microc/src/rtl.c

# --- Build feature1.list
nfcc -W3 -chip nfp-4xxx-b0 -Qspill=7 -Qnn_mode=1 -Qno_decl_volatile -
    single_dram_signal -Qnctx_mode=8 -FI config.h  -DBLM_CUSTOM_CONFIG
    -I. -I/dtree/mine/micro_c -I/c_packetprocessing/microc/include -I/
    c_packetprocessing/microc/lib -I/c_packetprocessing/microc/blocks/
    blm -I/c_packetprocessing/microc/blocks/blm/_h -I/components/
    standardlibrary/include -I/components/standardlibrary/microcode/
    include -Fefeature1_32.list /dtree/mine/micro_c/island32/
    feature1_32.c /c_packetprocessing/microc/lib/nfp/libnfp.c /
    c_packetprocessing/microc/lib/std/libstd.c /components/
    standardlibrary/microc/src/rtl.c

# --- Build feature2.list
```

```
nfcc -W3 -chip nfp-4xxx-b0 -Qspill=7 -Qnn_mode=1 -Qno_decl_volatile -
    single_dram_signal -Qnctx_mode=8 -FI config.h -DBLM_CUSTOM_CONFIG
    -I. -I/dtree/mine/micro_c -I/c_packetprocessing/microc/include -I/
    c_packetprocessing/microc/lib -I/c_packetprocessing/microc/blocks/
    blm -I/c_packetprocessing/microc/blocks/blm/_h -I/components/
    standardlibrary/include -I/components/standardlibrary/microcode/
    include -Fefeature2_32.list /dtree/mine/micro_c/island32/
    feature2_32.c /c_packetprocessing/microc/lib/nfp/libnfp.c /
    c_packetprocessing/microc/lib/std/libstd.c /components/
    standardlibrary/microc/src/rtl.c

... ### Repeat similar commands for remaining features ###

# --- Build result_collect.list
nfcc -W3 -chip nfp-4xxx-b0 -Qspill=7 -Qnn_mode=1 -Qno_decl_volatile -
    single_dram_signal -Qnctx_mode=8 -FI config.h -DBLM_CUSTOM_CONFIG
    -I. -I/dtree/mine/micro_c -I/c_packetprocessing/microc/include -I/
    c_packetprocessing/microc/lib -I/c_packetprocessing/microc/blocks/
    blm -I/c_packetprocessing/microc/blocks/blm/_h -I/components/
    standardlibrary/include -I/components/standardlibrary/microcode/
    include -Feresult_collect_32.list /dtree/mine/micro_c/island32/
    result_collect_32.c /c_packetprocessing/microc/lib/nfp/libnfp.c /
    c_packetprocessing/microc/lib/std/libstd.c /components/
    standardlibrary/microc/src/rtl.c

... ### Repeat similar commands for Islands 33-36 ###

# --- Link code
nfld -chip nfp-4xxx-b0 -mip -rtsyms -o model.fw -map model.map -u i32
    .me0 -l pkt_collect_32.list -u i32.me1 -l feature1_32.list -u i32.
    me2 -l feature2_32.list ... -u ila0.me0 -l blm.list -i i8 -e /
    components/standardlibrary/picocode/nfp6000/catamaran/catamaran.
    npfw

### Load Firmware onto SmartNIC ###
nfp-nffw load --no-start model.fw
nfp-nffw start
```