Taming the Elephants: Affordable Flow Length Prediction in the Data Plane

RAPHAEL AZORIN, Huawei Technologies Co. Ltd, France and EURECOM, France
ANDREA MONTERUBBIANO, University of Rome La Sapienza, Italy
GABRIELE CASTELLANO, Huawei Technologies Co. Ltd, France
MASSIMO GALLO, Huawei Technologies Co. Ltd, France
SALVATORE PONTARELLI, University of Rome La Sapienza, Italy
DARIO ROSSI, Huawei Technologies Co. Ltd, France

Machine Learning (ML) shows promising potential for enhancing networking tasks by providing early traffic predictions. However, implementing an ML-enabled system is a challenging task due to network devices limited resources. While previous works have shown the feasibility of running simple ML models in the data plane, integrating them into a practical end-to-end system is not an easy task. It requires addressing issues related to resource management and model maintenance to ensure that the performance improvement justifies the system overhead. In this work, we propose DUMBO, a versatile end-to-end system to generate and exploit early flow size predictions at line rate. Our system seamlessly integrates and maintains a simple ML model that offers early coarse-grain flow size prediction in the data plane. We evaluate the proposed system on flow scheduling, per-flow packet inter-arrival time distribution, and flow size estimation using real traffic traces, and perform experiments using an FPGA prototype running on an AMD(R)-Xilinx(R) Alveo U280 SmartNIC. Our results show that DUMBO outperforms traditional state-of-the-art approaches by equipping network devices data planes with a lightweight ML model. Code is available at https://github.com/cpt-harlock/DUMBO.

CCS Concepts: • **Networks** → **Network measurement**; • **Computing methodologies** → *Machine learning*; • **Hardware** → Reconfigurable logic and FPGAs.

Additional Key Words and Phrases: per-flow monitoring;in-network machine learning;data plane

ACM Reference Format:
Raphael Azorin, Andrea Monterubbiano, Gabriele Castellano, Massimo Gallo, Salvatore Pontarelli, and Dario Rossi. 2024. Taming the Elephants: Affordable Flow Length Prediction in the Data Plane. *Proc. ACM Netw.* 2, CoNEXT1, Article 5 (March 2024), 24 pages. https://doi.org/10.1145/3649473

1 INTRODUCTION

Flow size prediction is an important yet challenging problem in the networking community. Accurately predicting flow size provides valuable information for network administrators to improve network management, as acknowledged in [48]. Flow size distributions exhibit a heavy-tailed nature, with the majority of flows being short (referred to as *mice*) and a small fraction of them being

Authors' addresses: Raphael Azorin, raphael.azorin@huawei.com, Huawei Technologies Co. Ltd, Boulogne-Billancourt (Paris), France and EURECOM, Biot, France; Andrea Monterubbiano, monterubbiano@di.uniroma1.it, University of Rome La Sapienza, Rome, Italy; Gabriele Castellano, gabriele.castellano@huawei.com, Huawei Technologies Co. Ltd, Boulogne-Billancourt (Paris), France; Massimo Gallo, massimo.gallo@huawei.com, Huawei Technologies Co. Ltd, Boulogne-Billancourt (Paris), France; Salvatore Pontarelli, pontarelli@di.uniroma1.it, University of Rome La Sapienza, Rome, Italy; Dario Rossi, dario.rossi@huawei.com, Huawei Technologies Co. Ltd, Boulogne-Billancourt (Paris), France.

ACM 2834-5509/2024/3-ART5
https://doi.org/10.1145/3649473

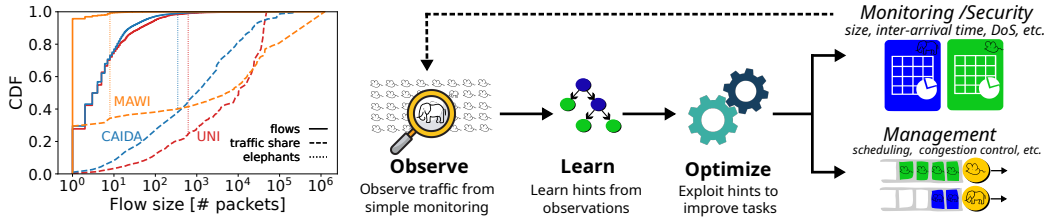


Fig. 1. (left) Flow sizes distribution and traffic share on three network traces. (right) Synopsis of DUMBO

significantly larger (known as *elephants*). To provide a concrete example, Figure 1 (left) reports the flow size distribution from three different traffic traces: CAIDA [2], MAWI [1], and UNI [14]. While elephants represent only a tiny fraction of all flows, e.g., the top 1%, they correspond to a substantial portion of the overall volume. Consequently, elephants significantly impact network management and monitoring tasks [26, 48]. Due to the dynamic nature of network traffic, precise flow size prediction is a difficult operation to perform, especially in the data plane.

We argue that early elephant flows identification is a more attainable objective, which still offers a significant advantage for network management and monitoring tasks. Our broad vision encompasses a networking system that seamlessly integrates machine learning (ML) models directly into the data plane, enabling the provision of timely flow size *hints*. Downstream networking tasks can then leverage these hints to enhance their overall performance and efficiency. One notable family of tasks that can greatly benefit from traffic predictions is *network management*: this includes congestion control, scheduling, and routing, wherein several studies have explored the usage of measurements or forecast-based hints [9, 24, 30, 32, 34, 39, 41, 43, 48]. Another family that would benefit from traffic predictions is *network monitoring*, in which approximate data structures are typically used to reduce system memory footprint. By leveraging ML hints, the monitoring system can dedicate more memory to elephant flows, which are naturally more important than the others cf. Figure 1 (left). In turn, this segregation helps reduce estimation errors, thereby improving the reliability of various monitoring tasks [29, 35, 50] as demonstrated by [22, 26].

A high-level view of an ML-enabled data plane pipeline is illustrated in Figure 1 (right). In a nutshell, the system collects lightweight observations (or measurements) which are used to train an ML model providing hints that are then exploited to improve downstream tasks. Despite its potential, several challenges must be addressed to realize such a system. One significant challenge pertains to the limited number of available operations and memory scarcity in network devices, e.g., SmartNICs, switches, etc. Introducing an ML model in network devices incurs overhead which effectively competes for resources with existing network functionalities. Therefore, the ML-enabled data plane benefits must outweigh the associated costs and apply to multiple network tasks, while ensuring that the imprecision of the model does not degrade the performance of the tasks it is supposed to support. This paper presents DUMBO, a comprehensive system that provides traffic characteristics hints in the form of simple binary classifications, elephants or mice flows. Our proposal is based on three key observations: *(i)* simple hints can be learned by a constrained machine learning model, *(ii)* whose implementation in the data plane can be realized with modern programmable hardware, and *(iii)* that holds significant value for multiple downstream networking tasks. While previous work has addressed some of these aspects individually [7, 22, 26, 49, 51], we conduct an end-to-end analysis of the system, spanning from design to implementation and experimentation. DUMBO leverages a single ML model to enhance various networking tasks, carefully considering the trade-offs between performance gains and overhead. By encompassing a holistic approach, we address the entire system lifecycle, examine its design choices, and evaluate

its benefits on the final tasks performance metrics. In contrast to existing literature, this approach enables us to accurately assess the capabilities and demonstrate the benefits of our system.

The paper is structured as follows, Section 2 introduces the scientific background and main motivations of this work, presenting the use cases we consider to showcase DUMBO namely flow scheduling, packet inter-arrival time distribution, and flow size estimation. Section 3 presents the system design and Section 4 investigates the development of a suitable ML model, including a thorough analysis of model performance, size, and update. Section 5 presents the details of the system's FPGA prototype. Section 6 validates the system end-to-end using real traffic traces, also providing a performance trade-off analysis. Section 7 concludes the paper. The code for model training and system evaluation is available at [4].

2 BACKGROUND AND MOTIVATION

In this section, we first motivate our ML-based data plane pipeline and position our contribution with respect to related work. We then introduce the three use cases we use as a reference to evaluate the effectiveness of our approach in an end-to-end fashion.

2.1 Machine Learning for networked systems

Predicting flow size. Previous studies have demonstrated the advantages of early flow size prediction in various networking scenarios, such as flow scheduling [48], routing [41, 43], congestion control [32], and flow measurements [22, 26]. While statistics-based heuristics can be used efficiently [13, 47], they often necessitate a long detection time, thereby diminishing the utility of the provided hints. Therefore, researchers have turned to ML techniques to train models that can approximate flow sizes in advance. Most studies have focused on offline analysis, proposing complex and computationally expensive Deep Learning models, such as Recurrent Neural Networks [22, 26].

The possibility of obtaining accurate flow size predictions has been questioned in [48], leading researchers to investigate the implications of acquiring such information. However, in the context of job scheduling, [37] recently provided theoretical evidence that a simple binary hint of whether a job is "big" or "small" can lead to significant gains in the scheduling task, even when the hint is not accurate. We believe that this observation is true also for many networking tasks that identify flows using TCP/IP five-tuple as described in Table 1. Hence, we consider a simple Random Forest model for a classification task, similarly to [22, 26, 51], i.e., predict "elephants" or "mice" flows based on the first few packets. We define elephants as the top-1% (cf. Figure 1–left) and mice as the rest. In [22, 26], authors only use features from the flow five-tuple, thus being able to perform classification using only the first packet. Because such an approach is highly vulnerable to traffic variations, pHeavy [51] proposes to wait for the first 5—20 packets to extract additional features, hence providing a later prediction. Compared to pHeavy, we limit DUMBO pipeline to the first 5 packets and develop a model that can be run at the beginning of the flow and is 3.5× more precise.

Deployment trade-off. Despite the rich literature, several challenges remain unresolved for practical flow size prediction in network devices data planes with a limited amount of resources e.g., Smart NICs, switches. We argue that prior works that address the ML deployment aspect of the problem [7, 8, 18, 46, 49, 51, 55] are far from demonstrating an end-to-end pipeline that can be

Table 1. Network use cases that can exploit elephants/mice hints for improved performance.

Family	Use case	References	Description
Network management	Congestion control	FACC [32], HPCC [34], ORCA [6], Swift [30]	*Elephant* flows are treated separately e.g., lower priority.
	Scheduling and routing	pHost [24], pFabric [9], MLRouting [41], Blaster [43],	
Security and Monitoring	Heavy hitter, DoS	Elastic Sketch [50] and CMS-like	*Elephant* flows are allocated more accurate data structures.
	Quantile estimation	KLL [29], DDSketch [35]	

adopted in practice. The traditional ML pipeline, involving the selection of the best model on a validation dataset and its subsequent use in the target environment, has known limitations [10, 11, 20]. This approach overlooks the properties of the target system and focuses solely on classification performance rather than evaluating the system end-to-end. Indeed, a critical aspect of data plane ML integration pertains to the trade-off between the model benefits and its deployment overhead. Although this trade-off has been partially studied [7, 8, 18, 49, 55], we posit that considerable effort is still needed. We highlight two main problems that affect the analysis done by existing works. On one hand, *(i)* the effectiveness of a model should be defined by its ability to consistently provide valuable hints for downstream tasks and the penalties that may result from incorrect predictions, and not by an offline performance metric. On the other hand, *(ii)* the operational overhead should be characterized by all the additional components needed for the model to operate in the data plane.

When designing DUMBO, we take into account these aspects and evaluate the deployment trade-off by accounting for the memory and processing overhead and relate the model performance metric with the actual end-to-end performance of the tasks. For the deployment to be practical, DUMBO should impose limited memory and processing overhead, and outperform its non-learned counterparts which operate without any hint but have access to more resources. Finally, we include in our pipeline a mechanism that periodically updates the model using newly sampled data to maintain the system performance stable over time. To the best of our knowledge, this is the first work proposing a complete data plane ML pipeline *(i)* detailing all its components, their deployment overhead, and a system prototype, *(ii)* evaluating the end-to-end pipeline performance on the downstream tasks, and *(iii)* demonstrating its long-term deployment effectiveness.

2.2 Use cases

Flow scheduling. Flow scheduling refers to the process of efficiently allocating network resources to different flows (e.g., identified by TCP/IP 5-tuple). It involves determining the next packet to be forwarded and possibly prioritizing some flows. Leveraging flow size estimation has emerged as a promising approach to optimize flow scheduling algorithms in recent years. For instance, pFabric [9] aims at minimizing the average flow completion time by prioritizing small flows over huge ones. However, it requires end-hosts to communicate flow residual length, which requires, in turn, applications and network devices modifications. An alternative strategy is pHost [24], which strives for optimal performance akin to pFabric, but without requiring network device modification. Unlike pHost and pFabric, DUMBO does not require the involvement of end-hosts to provide flow size hints. Yet, such a system could still prioritize smaller flows over larger ones, thereby offering scheduling performance similar to pFabric and pHost.

Flow IAT distribution estimation. A popular network measurement task for Service-Level Agreement is the estimation of packet inter-arrival time (IAT) quantiles. Precise IAT distribution monitoring is impractical given the amount of memory it would require, which calls for approximate measurement techniques. A well-known sketch for distribution estimation is DDSketch [35], which is allocated once per each flow under measurement. A DDSketch contains multiple buckets (more buckets lead to more accurate estimation) that correspond to different ranges of IAT values. Mice are composed of a few packets, which yield only a handful of IAT values to be inserted into the corresponding DDSketch buckets, thus leaving the majority of the buckets empty. Early knowledge of flow size can help dimension the number and the size of buckets to allocate per flow. Therefore, DUMBO employs flow size hints to use more and/or bigger buckets for elephants. To the best of our knowledge, we are the first to propose a learned DDSketch using flow size hints. Other sketches such as KLL [29], and GKsketch [25] can similarly benefit from flow size hints.

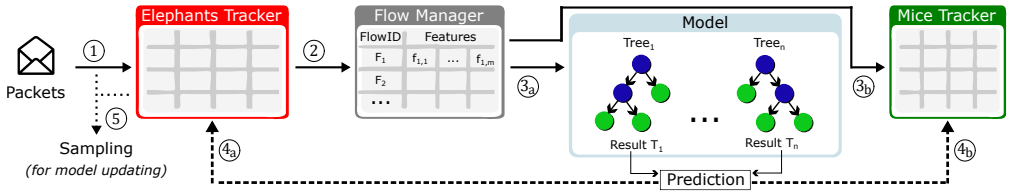


Fig. 2. DUMBO system architecture.

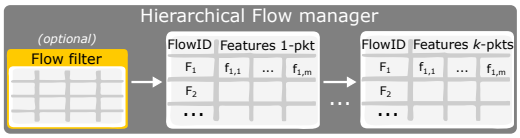
Flow size estimation. Flow size estimation is a popular network monitoring task that is typically addressed by employing probabilistic data structures like the Count-Min Sketch (CMS) [19]. In a CMS, flow keys are hashed to identify flow counters that are shared across multiple flows. Intuitively, the estimation error of mice flows that share counters with elephant flows is particularly impacted. Recent variants of the CMS attempt to segregate elephants from mice and count them in separate data structures. For instance, ElasticSketch (ES) [50] uses a voting algorithm to decide whether a flow should be counted in the CMS or in a dedicated bucket. Parallel to this line of research, [22, 26] explore the use of an ML model to make such a decision. These works show promising results evaluating the learned CMS offline, yet they do not take system implementation into account.

3 SYSTEM DESIGN

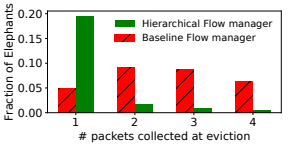
Our system is built around the concept that coarse flow size predictions in the data plane benefit several network tasks. For this reason, we aim at classifying flows based on their size, with reasonable confidence, and as early as possible within the first few packets of the flow. Late predictions would reduce the benefits of the ML-enabled data plane pipeline. In this section, we present the system design for integrating such ML hints in the data plane, while we defer to Section 4 the details of the ML model design. Figure 2 presents the high-level DUMBO system design with its four components: *Elephant Tracker*, *Flow Manager*, *Model*, and *Mice Tracker*. In the following, we first introduce the processing pipeline, and then we detail the specific design of each component.

3.1 Pipeline overview

The first component in the processing pipeline (cf. Figure 2) is the *Elephant Tracker*, whose role is twofold. First, it collects precise monitoring data (e.g., packet count, timestamps, etc.) for the elephant flows. In addition to that, it serves as a cache for elephant predictions from the model. Upon receiving a packet (step ①), the system performs a lookup in the Elephant Tracker. If the flow is found therein, i.e., the flow was already predicted as an elephant, it is updated, and the processing pipeline ends. Otherwise, the packet is sent to the *Flow Manager* (step ②), which detects if the packet belongs to a new flow, e.g., using the SYN flag or relying on a bloom filter [15]. The primary role of the Flow Manager is to collect the features that need to be extracted from the first k packets of each flow. Once these features have been collected, they are sent to the *Model* (step ③*a*), triggering a prediction. If the Model predicts an elephant flow, the packet is sent to the Elephant Tracker, e.g., by packet recirculation in modern programmable pipelines, together with the information collected so far, and a new entry for this flow is created (step ④*a*). If the model predicts a mouse, the packet is sent to the *Mice Tracker* component instead (step ④*b*), where monitored metrics are stored in a compact probabilistic data structure, e.g., with sketches. In addition, the Flow Manager serves as a negative cache for flows predicted as mice: if a packet that does not belong to a new flow, and has no entry in the Elephant Tracker nor in the Flow Manager, it is directly sent to the Mice Tracker (step ③*b*). Similarly, any entry that gets evicted from the Flow Manager before collecting k packets is directly treated as a mouse without triggering any prediction. Note that the system continuously samples packets to trigger model updates when needed (step ⑤).



(a) Architecture of a hierarchical Flow Manager. Flow entries move to the next stage when a new packet is collected.



(b) Simple vs. Hierarchical Flow Manager elephants evictions ($k = 5$ packets).

Fig. 3. Hierarchical Flow Manager design (a) and elephants discarded before reaching $k = 5$ packets (b).

3.2 System components

Elephant Tracker. The Elephant Tracker keeps track of flows that are predicted as elephants by the model. It can be implemented with a simple multi-level[1] and multi-entry[2] hash table. This hash table stores 6-byte flow fingerprints[3] and precise flow monitoring data for the task at hand (e.g., 3-byte counters recording flow length). Hash table and fingerprint collisions can be handled using existing Flow to ID mapping solutions such as [12, 40, 44, 52] that fit in modern programmable hardware like FPGA-based smart NICs and Tofino switches.

The Elephant Tracker is accessed whenever a new incoming packet is processed by the system. We first extract the flow fingerprint from the packet, then perform a lookup on the hash table for the flow entry, that is, we check if the flow was already classified as an elephant by the Model. If so, the data stored in the entry is updated according to the metrics extracted from the last packet (e.g., increment the corresponding counter recording flow length). The data stored in each flow entry might significantly vary based on the monitoring tasks performed by the system. For instance, in the case of measuring packets IAT, each entry of the Elephant Tracker features the timestamp of the last seen packet and a DDSketch to provide accurate quantile estimation. In this use case, the system updates the latest timestamp and uses the previous one to compute the IAT, then it updates the DDSketch accordingly. Insertions of new entries in the Elephant Tracker are dictated by the Model, i.e., the system occupies a pre-reserved entry whenever a new flow is predicted to be an elephant. Notice that, due to the use of a hash table, such insertion may fail (that is, the entry in the hash table is already allocated to another flow). In this case, the system treats the new flow as a mouse and forwards its packets and metadata to the Mice Tracker. We note that the Model decision needs to be back-propagated to the Elephant Tracker; however, this operation is not possible in programmable switches, because the Elephant Tracker is the first component of the processing pipeline. Therefore, we need to recirculate a certain fraction of packets. Since this is only needed at insertion time, i.e., one packet per inserted flow, and since elephant flows account for only 1% of all the flows, we expect a negligible recirculation overhead due to the Elephant Tracker.

Flow Manager. The Flow Manager is responsible for collecting flow features from the first k packets of each flow that are necessary to get hints from the Model. Once the required features have been accumulated, the Flow Manager forwards them to the Model for inference. Similarly to the Elephant Tracker, the Flow Manager is organized as a multi-entry hash table, using the same 6-byte fingerprint as a key. The Flow Manager is accessed only when there is a miss in the Elephant Tracker. If this is the case, the system first checks if the packet belongs to a flow that has never been seen so far. To do so, a possible strategy is to exploit packet headers in case of TCP flows i.e., SYN, SYN-ACK flags. Alternatively, to accommodate non-TCP traffic, another strategy is to query an optional *Flow Filter* component as illustrated in Figure 3a, which can be implemented with a Bloom Filter [15]. If the flow is new, it is inserted in the hash table in a free bucket or evicting an old flow.

[1]Multi-level refers to a hash table with h successive hash functions used to address h separate tables.
[2]Multi-entry refers to a hash table with multiple entries for a single addressable bucket.
[3]Based on the use case, an efficient fingerprinting algorithm can be derived by simply hashing the 5-tuple flow identifier.

The new entry is initialized with the features extracted from the first packet and its timestamp. If the flow is not new and there is a match in the Flow Manager, the corresponding entry is updated with the latest feature values. When the k-th packet of the flow arrives, the system packages the collected features from the updated counters values, extracts the 5-tuple features from the packet header (TCP/IP ports and addresses), and forwards them to the Model for inference, also freeing the entry in the Flow Manager. Finally, if the flow is not new but is not in the Flow Manager, it is considered a mouse and directly forwarded to the Mice Tracker for further processing. This situation can occur in two cases: *(i)* the flow has been previously predicted as a mouse by the model and then evicted from the Flow Manager, or *(ii)* the flow has been evicted from the Flow Manager before reaching k packets. We remark that flow eviction occurs when there is a collision in the hash table and no free slots are available to store the new flow. In this case, we evict the least recently used flow entry in the bucket. In practice, after detecting the entry with the smallest timestamp within the bucket and its position, its place is taken by the newly inserted flow: the flow evicted in this process is sent to the Mice Tracker and considered as mouse thereupon.[4]

The Flow Manager described above has the drawback of possibly evicting flows close to reaching k packets, i.e., just before model inference. To address this problem we organize the Flow Manager hierarchically as illustrated in Figure 3a. The first table keeps track of flows that have accumulated one packet so far, the second table those that have accumulated two packets, and so on. Eviction is restricted among flows with the same amount of packets. In this way, flows close to reaching k packets are implicitly given priority over the others. Note that this hierarchical structure is compatible with programmable pipelines because when a hit occurs, the flow simply needs to be moved to the next level. Figure 3b compares a hierarchical Flow Manager with a simple one, using a CAIDA traffic trace with $k = 5$ and a fixed memory budget. In this simple example, the hierarchical version reduces early evictions of the elephants to 22% (32% with simple Flow Manager).

Model. The model is responsible for producing hints in the form of a binary elephant/mice classification by taking as input the flow features collected by the Flow Manager. We recall that, if a flow is evicted from the Flow Manager before accumulating k packets, it will never get to the Model and thus never get a chance to be correctly classified. Therefore, the Flow Manager plays a key role in the overall quality of the hints provided by the Model. Hence, we highlight the importance of carefully co-designing the Flow Manager and Model components to get the best performance.

Mice Tracker. The role of the Mice Tracker is to approximately monitor flows predicted as mice. Simply disregarding these flows would be inaccurate because some of them are actually elephants, while others did not get a model prediction due to early eviction from the Flow Manager without even receiving a model prediction. As a result, the Mice Tracker serves as a backup monitoring data structure, similar to the approach used in [26, 50]. In our system, the Mice Tracker comprises one or multiple approximate data structures such as Count-Min Sketch for flow size estimation or compact DDSketches [35] for IAT distribution estimation.

4 MACHINE LEARNING MODEL

The goal of the DUMBO ML model is to classify flows according to their size as quickly as possible while maintaining high performance and low memory overhead. In this section, we first introduce the metrics and baselines used to compare our approach, and then we detail model design and sizing. Finally, we present an automatic update mechanism to ensure model robustness over time.

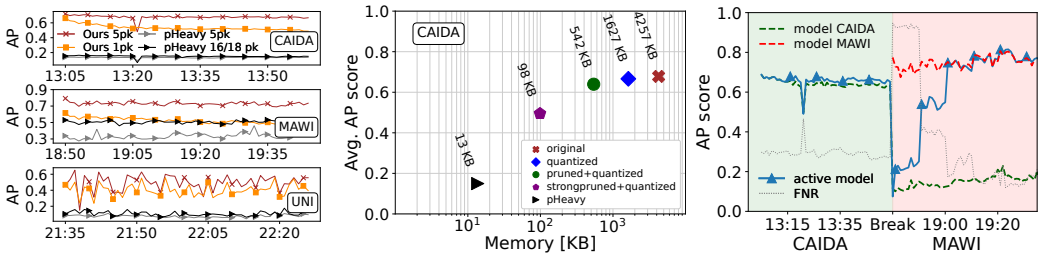
[4]This can be done without recirculation in FPGAs by adopting a two-stage pipelined memory access: the first stage for reading the memory, the second for performing the update. This approach would be hard to implement in other popular programmable architecture, like Intel(R) Tofino, due to its constrained memory model, hence requiring packet recirculation.

Benchmark setup. To motivate our choices, we run model micro-benchmarks using CAIDA [2] (2016-01-21 13:00–13:59), MAWI [1] (2019-04-09 18:45–19:44), and UNI (2010-01-12 20:00—22:29) [14] traffic traces. As [22, 26], we label flow sizes above the 99th percentile as elephants (positive class) and the rest as mice (negative class). For the evaluation, we focus on TCP and UDP flows and deliberately exclude ICMP traffic that would make the prediction task less challenging (cf. ICMP results in Appendix A). We consider minute-defined measurement epochs, which represent ≈1M, ≈500K, and ≈5K uni-directional flows for CAIDA, MAWI, and UNI respectively. The elephants/mice classification task is heavily imbalanced by design resulting in unequal consequences for mispredictions. It is important to note that binary classifiers typically output the *probability* that a sample belongs to the positive class. The classification result depends on a fixed threshold applied to this output. Tuning this probability threshold corresponds to selecting a trade-off between Precision and Recall. As a reminder, precision P and recall R take their values in the interval $[0, 1]$ (higher is better) and are defined as $P = T_p/(T_p + F_p)$, $R = T_p/(T_p + F_n)$ where T_p are true positives, F_p false positives and F_n false negatives. To ensure a fair comparison between classifiers, an appropriate metric is the Average Precision (*AP*) score, $\text{AP} = \sum_n (R_n - R_{n-1}) \cdot P_n$ where P_n and R_n are the precision and recall for the n-th threshold. In a nutshell, the AP score is the average precision at each threshold, weighted by the increase in recall from the previous threshold, and summarizes the trade-offs achievable with a given model by considering all its possible probability thresholds. In our benchmark campaign, we compare against pHeavy [51] that uses a pipeline of successive decision trees queried at various packet arrivals (e.g., 5-20) to filter out mice progressively. Hence, pHeavy can only predict elephants when the last model stage is reached (e.g., 20^{th} packet).

4.1 Model design

We choose Random Forests (RF) [17] models which are typically preferred over more sophisticated algorithms (e.g., Deep Learning) for data plane implementation, due to their simplicity and interpretability [7, 18, 33, 49, 53–55]. Ideally, the RF model should provide a prediction as early as possible i.e., at the first packet. In this case, the flow 5-tuple is the only feature fed to the model. As in [22, 26], the 5-tuple is transformed into 97 binary features: 32 + 32 features for source and destination IPs, 16 + 16 features for source and destination ports and 1 feature for protocol (TCP or UDP in our case). Binary features have the advantage of simplifying the RF deployment in the data plane. Similarly, it is also desirable for a data plane implementation to have binary leaves only. In vanilla RF, each leaf is assigned an impurity value, which in practice equals the proportion of one class over the other among training samples that map to that leaf. Such values are provided as output by every tree at inference time and then averaged across the forest to output a class probability. This probability of the sample belonging to the positive class is then thresholded to output the final elephant vs mouse classification result. To simplify data plane implementation and avoid costly floating point operations, we fine-tune an appropriate probability threshold within the training pipeline, encode it directly in each tree leaf, and use them as votes for class prediction.

While obtaining timely predictions from the first packet is attractive, this can limit the model's long-term performance. Therefore, we integrate additional flow features to better characterize their behavior. After analysis (not presented here for lack of space), we select four *aggregated features* collected from the first k packets of the flow, namely the *mean* and *standard deviation* of *packet sizes* and *inter-arrival times*. To do so, the Flow Manager stores a 2-byte timestamp of the last packet and four 2-byte counters for the aggregated features per each flow. These counters store the sum of the measured values $S = \sum_i x_i$ and the sum of their square $Q = \sum_i {x_i}^2$. After k packets, mean and standard deviation are computed as $\mu = S/k$, and $\sigma = \sqrt{(kQ - S^2)/(k(k-1))}$. Overall, the k-packets model has 101 input features: 97 binary features for the 5-tuple and 4 aggregated features.



(a) Model performance across time. (b) Model performance against size. (c) Model update stress-test.

Fig. 4. Machine Learning model benchmark.

To validate our model design, we train the Random Forest on the first 5 minutes of traffic in CAIDA, MAWI, and the first 95 minutes in UNI (as this trace features fewer flows), and use the last 55 minutes for testing. The model hyper-parameters (number of trees, max depth, etc.) are obtained through random search with 2-fold cross-validation on the training minutes, optimizing the Average Precision (AP) score. This first training phase does not consider any model size constraint; we consider this model as an upper bound in terms of achievable performance. Figure 4a compares models' performance on the test set constituted by the remaining 55 minutes of each trace. We remark that, as expected, adding aggregated features (with $k = 5$) leads to stable performance within the full test set. We also tested $k > 5$ but obtained stable performance starting from $k = 5$, hence, in the remainder of this paper we use the 5-packet model. We remark that using only aggregated features (no 5-tuple) does not provide accurate predictions (AP score ≈20% lower, not reported in the figure) as the model cannot correlate flow characteristics with its source and destination. Finally, we remark that classification on MAWI appears easier than on the two other traffic traces. This is partly explained by their different flow size distributions: in MAWI ≈90% of the flows have less than five packets, while this represents only ≈50% of the flows in CAIDA and UNI. This also explains the higher performance of pHeavy on MAWI: when using its last pipeline stage at the 18^{th} packet arrival in the model we trained, most mice have already been filtered out. The results on UNI indicate comparatively lower and less stable performance. This can be attributed to the limited amount of flows available for training and testing, respectively (i.e., ×100 less than CAIDA). To finalize our model for the ML-enabled data plane pipeline integration, we then use the last minute of the training set to empirically tune the probability threshold for elephants prediction. This threshold has a direct impact on the model Recall and Precision: the lower the threshold, the larger the number of flows predicted as elephants thereby favoring Recall over Precision and vice versa. Interestingly, this threshold enables us to control the fraction of flows that the model classifies as elephants, which is a crucial parameter for appropriately sizing the Elephant Tracker. Once the appropriate threshold is selected, it is encoded in all trees' leaves.

To interpret the trained model, we use the TrusteeML library [28] to extract a high-fidelity and low-complexity tree that explains the decisions of the model. TrusteeML is a framework that aims at identifying model under-specification issues such as shortcut learning or spurious correlations. It takes a model-agnostic approach to iteratively train various decision trees (students) that best mimic the model's decisions (teacher). The student with the highest fidelity is selected for top-k branch pruning to make it easily interpretable at the expense of performance. This process is repeated several times to select as final explainer the pruned student that is the most stable across runs (i.e., the highest mean agreement across pruned students). Due to lack of space, we report this simplified and interpretable decision tree in Appendix A. When analyzing the model, we note that the four aggregated features we introduce rank first in terms of feature importance (measured by the number of samples the feature is used to classify).

4.2 Model size

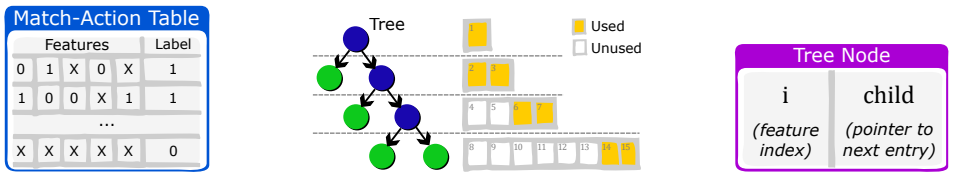
In the data plane, memory is often limited, making it crucial to limit the size of the model size to minimize system overhead. Alternative approaches for elephant flow detection, such as pHeavy [51], employ tiny models but require ≈5 MB per 100K flows to store the required features because of late predictions e.g., at 20^{th} packet. We argue that reserving this amount of memory is problematic in programmable network devices. Data structures dedicated to flow-size monitoring and management are typically allocated less than 1 MB of memory. For this reason, and to limit the system overhead, we target much smaller memory footprints for the full system, i.e., in the order of 1 MB. To evaluate the amount of memory required for the RF, as in [38] we consider three alternative decision tree implementation approaches: *Match Action Table* (MAT), which exploits MAT available in programmable network devices, *full tree*, which involves storing fully-grown binary trees (including nodes not used by the actual decision tree), and *hybrid*, which consists in storing fully-grown trees up to a certain depth, and individual tree nodes from that level downward. We defer the reader to Section 5 for additional details about model system implementation and size estimation. Here we report only the model memory occupation obtained using the best strategy among the three.

As depicted in Figure 4b, the original 5-packets model we obtain by training on the first 5 minutes of the CAIDA trace requires over 4 MB. It achieves an average AP score of 0.68 in the remaining 55 test minutes. This is influenced by unconstrained model parameters such as *(i)* the memory used to store feature values, and *(ii)* the number of trees composing the forest. To reduce the model size and meet the memory constraints of the data plane, we employ simple strategies with training speed and future model updates in mind. These strategies include quantization of feature values and forest pruning. For *(i)*, we discretize the aggregated features using quantiles. This pre-processing step is part of the automated model training process. It creates an input feature vector with only binary values, making it easier to integrate the model into the data plane. Aggregated features quantization provides a ×2.6 reduction in size for a 1.7% decrease in performance. For *(ii)*, we prune the forest by removing the worst-performing trees until a given size constraint is respected (≈500 KB in our settings). This procedure allows precise control over the size-performance trade-off of the final model and is also part of the model training routine. We select a conservative final model of size 542 KB (over ×7.8 size reduction) for an AP score of 0.64 (5.7% performance decrease).[5] We observe similar model size reduction on MAWI and UNI, not reported here for lack of space.

4.3 Model update

Model maintenance is needed to account for drifts in traffic behaviors and ensure that the quality of the predictions is sufficient for benefiting downstream tasks. Since traffic patterns are dynamic, we expect model performance to degrade over time, for which we need to periodically update the model. In our case, the classification task requires new training samples i.e., input features describing the flow and the corresponding label, to learn and adapt to the newest patterns. The challenge lies in the sampling policy implemented to acquire fresh data points. On the one hand, we should not distort the flow size distribution to keep the training set representative of the real data. On the other hand, we need to constrain sampling to incur a reasonably low overhead. Following these guidelines, we draw from active learning and uncertainty sampling [21, 45] to determine which flows to sample. During each measurement epoch, the model itself is used to identify challenging flows that should be sampled. By aggregating votes from individual RF trees, we can quantify the uncertainty of the model prediction. Hence, we select for sampling flows with more than $k = 5$ packets that do not reach a voting agreement threshold e.g., 0.6 and send them to the control plane. It is important to note that the agreement threshold is different than the one for model prediction.

[5]Stronger pruning led to a 98 KB model with a 0.50 AP score; we stick with the 542 KB model to provide conservative results.



(a) MAT tree optimized encoding. (b) Full tree encoding. (c) Indexed encoding (node).

Fig. 5. Visualization of different techniques for encoding a binary tree in the data plane.

This sampling strategy has the advantage of selecting challenging flows with limited overhead, but it may also distort the actual data distribution. To alleviate this shortcoming, we additionally randomly sample 1% of the flows that reach the model, i.e., 1% of the flows with more than $k = 5$ packets. In our experiments, combining both uncertainty and random sampling leads to an overall sampling rate below 3% in the worst case. Sampled flows collected at the controller are appended to the initial training dataset to constitute a retraining buffer. To keep a stable re-training cost, the training buffer is a FIFO queue of a fixed size equal to the initial training dataset (e.g., 2.5M flows for CAIDA). Finally, we implement a drift detector to adapt quickly to sudden traffic changes. This detector triggers when the flow features sampled by the controller drift abruptly from one epoch to another, resets the retraining buffer, and triggers frequent model updates.

Starting from a model trained on CAIDA, Figure 4c shows a 10-minute periodic active model update strategy in a worst-case scenario where, after 50 minutes of CAIDA test traffic (corresponding to a slow drift), the traffic abruptly changes to MAWI for the next 50 minutes (corresponding to out-of-distribution). Clearly, due to significant differences (e.g., IAT, IP addresses, flow size distribution), we expect a static model trained on CAIDA to be unfit to perform flow classification on MAWI traffic (dashed green line). However, even in such an extreme scenario, the active model update strategy (solid blue) quickly recovers to match the performance of a static model trained exclusively on MAWI data (dashed red). We also report FNR over time, i.e., the rate of elephants mispredicted as mice, and show that it only requires two model updates to return to reasonable levels. We remark that model re-training does not need to be frequent and periodic, but can be triggered when performance on randomly sampled flows becomes unsatisfactory (i.e., after a long drift or an abrupt change). Finally, we highlight that the training time for a single tree on such a large dataset is ≈30 s on a single core from an Intel(R) Xeon(R) Platinum 8164 2.00GHz CPU (≈1 minute training with 15 cores for a forest composed of 30 trees). Subsequent updates of the model weights on the FPGA target do not introduce any considerable extra overhead (≈1 ms in our case for rewriting the trees in the FPGA memory), hence allowing for line-rate updates in our system.

5 SYSTEM IMPLEMENTATION

The introduction of SmartNICs and programmable switches has opened up opportunities to offload an increasing number of network applications from end-host CPUs (precious to cloud providers) to networking devices. While programmable switches could potentially incorporate the proposed ML-enabled data plane, we believe that FPGA-based SmartNICs are ideal candidates for implementing DUMBO due to their rapid adoption and flexibility. The first public example of the deployment of FPGA-based SmartNICs was presented in 2018 [23], where a fleet of more than 1 million hosts is equipped with an ad-hoc SmartNIC. Today, key players such as Alibaba, Amazon, Huawei, and Google expose FPGAs to application developers in their data centers. A comprehensive survey discussing these and other advances [27, 36, 40, 42] can be found in [16]. In this section, we first briefly describe the way we embed a Random Forest in the FPGA, and then we present the full system implementation and evaluation on the AMD-Xilinx Alveo U280 card.

5.1 Random Forest FPGA Implementation

When implementing an RF in a programmable pipeline, i.e., FPGA in our case, it is critical to meet stringent throughput and memory size constraints. In particular, achieving one inference at each clock cycle is desirable for packet processing tasks as it greatly simplifies implementation. This calls for a processing pipeline that can be realized using a *MAT*, *full tree*, or *hybrid* representation as in [38]. As mentioned in Section 4, we restrict our analysis to binary leaves and features.

MAT. MATs are commonly employed in network devices. Packet-processing tasks are implemented by a sequence of MATs, each operating based on the result of the previous one. In [33] MATs were used to implement a decision tree with each level of the tree represented by a match-action stage. Alternatively, [49] encoded the entire tree using a single MAT, and [55] extends this principle to Random Forests. In a nutshell, each tree in the RF requires one MAT (+1 in some particular cases). The memory required (in number of bits) for a *RF* implemented with MATs is then:

$$Mem_{MAT} = \sum_{t \in RF} L_t (2\,F + 1), \tag{1}$$

where t represents a tree from the Random Forest, L_t is the number of leaves for t, and F is the number of binary features. The factor of two relates to the encoding of a ternary value (i.e., "0", "1", or "do not care") for each feature, that requires two bits. A simple optimization consists of encoding only the leaves corresponding to one of the two classes. When one of the two classes is under-represented within the tree's leaves, this leads to a decrease in memory, since only the leaves that correspond to the unfrequent class are saved in the MAT. We depict this approach in Figure 5a.

Full tree. In contrast to the MAT implementation, the full tree approach reserves a separate memory block for each level of the trees. Each node in the tree is numbered sequentially starting from 1 to $2^{D_t} - 1$, where D_t denotes the tree depth, as depicted in Figure 5b. Consequently, traversing the tree for inference is simple: given any node i, its children at the next layer are found at the memory offset $2i$ and $2i + 1$. However, this approach potentially leads to considerable memory overhead if the tree is sparse as it assumes each tree to be fully grown. Each node stores the feature index or predicted class label, as well as a bit to differentiate between split or leaf nodes, resulting in the following memory requirement (in number of bits):

$$Mem_{FT} = \sum_{t \in RF} (2^{D_t} - 1)(\lceil log_2(F) \rceil + 1), \tag{2}$$

This representation can be easily implemented in an FPGA pipeline where each stage is responsible for accessing one layer of the tree. Layers with few nodes can be implemented using LUTRAM or Flip-Flops, while larger blocks would require to be mapped to BRAMs.

Hybrid. In our case, we observe that the trees composing the RF are often full only up to a certain level; hence, the full tree implementation leads to significant memory waste for the deepest layers that contain very few nodes (e.g., last layer in Figure 5b). We propose a hybrid approach that uses the full tree representation for the top layers and an *indexed encoding* representation for the bottom ones. We denote the maximum number of nodes in any layer of any tree in the forest by N. We encode the top $M = \lfloor log_2(N) \rfloor + 1$ layers using the full tree implementation. For the remaining layers, we allocate only N nodes. For each of these nodes, we store a pointer to their left child located in the next layer (cf. Figure 5c): the right child is thus located at this address plus 1. The memory required (in number of bits) for a hybrid *RF* representation can be expressed as:

$$Mem_{Hybrid} = \sum_{t \in RF} \underbrace{(2^M - 1)(\lceil log_2(F) \rceil + 1)}_{\text{full tree implementation}} + \underbrace{N(D_t - M)(\lceil log_2(F) \rceil + 1 + \lceil log_2(N) \rceil)}_{\text{indexed encoding}}, \tag{3}$$

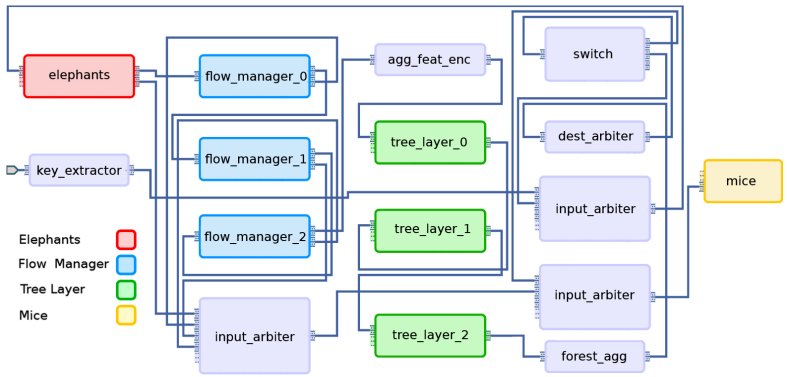


Fig. 6. Xilinx Vivado block diagram of the DUMBO pipeline for a simplified 3-layer RF model

5.2 Full system implementation

We realize a fully working prototype wrapping the DUMBO architecture depicted in Figure 2 into the Xilinx OpenNIC shell [3], providing a system able to process a 100 GbE link. We consider the flow size estimation use case, where the Mice Tracker is a Count-Min Sketch (CMS) with 4×2^{14} buckets of 2 bytes and the Elephant Tracker is a hash table with 4×2^{12} slots containing exact counters. The Flow Manager is composed of a total of 2^{15} slots. For the RF hybrid implementation, we implement a model composed of 33 trees with a maximum depth $max_{t \in RF}(D_t) = 23$ levels and a maximum number of nodes per layer of $N = 940$. Thus, we fix the threshold to switch from the full tree implementation to the indexed encoding at $M = 10$. All the syntheses have been carried out targeting the AMD-Xilinx Alveo U280 and an operating frequency of 180 MHz, which is more than the one needed to sustain the full throughput of 144 Mpps, *i.e* the maximum rate for a 100 GbE link. Figure 6 depicts the main DUMBO building blocks where for the sake of clarity, we show a 3-layer tree instead of the full RF model. Packets flow from left to right, neglecting the light grey blocks which contain minor glue logic. First, packets traverse the Elephant Tracker, then three levels of Flow Manager, three layers of decision tree, and are finally directed to the Mice Tracker or the Elephant Tracker based on the prediction provided by the model. Finally, packet and byte counters are disseminated into the OpenNIC shell to monitor the datapath and identify the location of possible packet drops.

Table 2 reports the FPGA resource consumption of the full system and of the DUMBO pipeline (neglecting the OpenNIC shell). The full system requires less than 15% of available logic resources and ≈40% of memory resources. Note that a significant fraction of logic resources is due to the fixed overhead of the OpenNIC infrastructure. Instead, if needed, DUMBO resources can be augmented in terms of memory for Elephant and Mice Trackers and a larger RF Model. From a performance point of view, we measure FPGA throughput and latency. The full system is able to sustain the full 100 GbE throughput regardless of packet size. To assess the latency of the prototype, we test both the full system included in the shell and the plain OpenNIC shell without the DUMBO pipeline. We report that the baseline OpenNIC shell has a latency of around 960 ns. For the whole system, as we consider the flow size estimation use case which is a pure monitor application, the forwarding decision is taken independently from the counting procedure. Thus, there is no latency degradation due to the ML pipeline. If we consider the scheduling use case instead, the packet is classified after traversing the Elephant Tracker and Flow Manager blocks, inducing an additional latency of 14 clock cycles (≈70 ns).

Table 2. Prototype resources utilization.

	DUMBO	Full system
CLB LUTs	93878 (7.20%)	187419 (14.38%)
CLB Registers	242467 (9.30%)	369083 (14.16%)
Block RAM	717 (35.57%)	860.5 (42.68%)

Table 3. Memory allocation for DUMBO.

Component	System	Scheduling	IAT est.	Flow size est.
Model	542 KB	n.a.	n.a.	n.a.
Flow Manager	290 KB	n.a.	n.a.	n.a.
Elephant Tr.	117 KB	n.a.	1.26 MB	59 KB
Mice Tr.	n.a.	n.a.	Remaining mem.	Remaining mem.
TOTAL	**1 MB**	**n.a.**	**33 MB**	**2 MB**

6 EXPERIMENTAL RESULTS

In this section, we detail the memory allocation for the system components and evaluate DUMBO on three tasks: flow scheduling, flow IAT distribution estimation, and flow size estimation. First, for every use case, we compare the end-to-end performance of DUMBO with state-of-the-art baselines and with a competitor ML pipeline based on pHeavy hints [51] obtained within the first 5-packets. Then, we analyze the deployment trade-off between model performance and pipeline memory overhead. We run our tests on packet-level simulators using the last ten minutes of traffic traces from CAIDA, MAWI, and UNI datasets. We highlight that UNI features ≈5K flows per minute, which is significantly lower than CAIDA and MAWI with ≈1M and ≈500K flows per minute, respectively. However, we still include results on UNI, also used in [51], as it enables us to estimate the applicability of our approach on datacenter-like traffic.

6.1 Memory setup

We categorize memory allocation into *system*-related, and *task*-related. This distinction stems from the observation that, while system-related memory is shared across the various network tasks that benefit from ML hints, task-related memory may significantly vary based on the use case.

System-related. In DUMBO we allocate 1 MB in total for Model, Flow Manager, and Elephant Tracker. We consider a conservative model size of 542 KB. The size of the Flow Manager depends on the number of simultaneous flows. We use the CAIDA training traces, which feature the highest amount of flows across our evaluation traces, to estimate this quantity. Accordingly, we allocate a hierarchical Flow Manager whose stages are broken down following the observed flow size distribution; this amounts to 290 KB, each entry requiring 6 bytes for the flow fingerprint, 2 bytes for the timestamp, and four 2-byte counters to collect the aggregated features. The size of the Elephant Tracker depends on the number of flows predicted as elephants, which is controlled in DUMBO by tuning the model probability threshold. The training pipeline uses the last minute of the training trace to empirically determine a threshold that sends 2% of the flows to the Elephant Tracker (i.e., ≈20K flows on CAIDA with a 20% overprovisioning). This amounts to 117 KB, each entry requiring 6 bytes for the flow fingerprint.

Our implementation of the pHeavy model only takes 13 KB. However, flow management requires much more memory due to extra features and eviction strategy; based on [51], it would take ≈50 MB on CAIDA traces with 1M flow per minute. To make the comparison feasible, we replace flow tuples with our same fingerprints and apply a more aggressive eviction strategy that replaces flows after 5 seconds of inactivity, thus constraining pHeavy flow management to ≈1.6 MB in the worst case. For the Elephant Tracker, we use the same configuration as in DUMBO and similarly tune a model probability threshold.

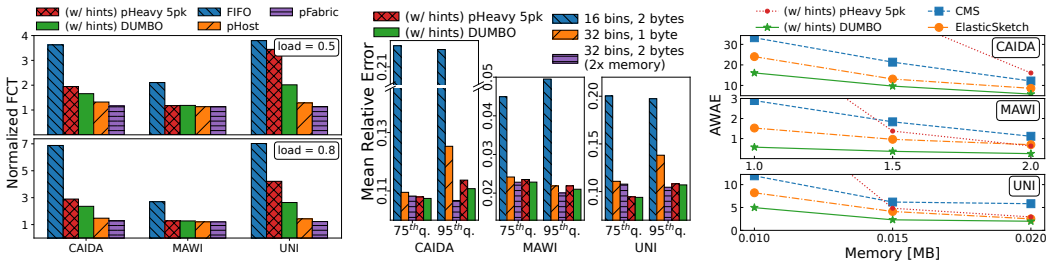
Task-related. The base system is sufficient to run the scheduling use case. The two other use cases (IAT distribution and flow size estimation) however, require additional memory for measurements for both Mice and Elephant Trackers. In our experiments, we allocate 33 MB of memory for IAT distribution and 2 MB for flow size estimation. Detailed memory allocations can be found in Table 3.

In particular, the Elephant Tracker for flow size estimation requires one additional 3-byte counter for measurements, while the IAT distribution estimation use case requires a 2-byte timestamp and a DDSketch. Finally, the remaining available memory from the use case budget is fully allocated to the Mice Tracker. This consists of a Count-Min Sketch for flow size estimation or an array of coarse DDSketches for IAT distribution estimation. In the case of pHeavy, we remove from this remaining budget the extra memory it needs for flow management.

6.2 End-to-end performance

Flow scheduling. For the flow scheduling use case, we evaluate the performance of the system in terms of average normalized Flow Completion Time (FCT) as defined in [9, 24], i.e., $normalized\ FCT = actual\ FCT / ideal\ FCT$. The ideal FCT is the time required to transmit a flow when the whole network fabric is composed of a single 10 Gbps link and there are no other flows, while the actual FCT is the measured flow completion time. We design a scheduling policy based on model predictions, assuming that the Flow Manager collects measurements from the first $k = 5$ packets of each flow. Each packet is assigned one out of three priorities from 0 (highest) to 2 (lowest). We assign priorities as follows: if the packet is one of the first $k - 1$, it is assigned to the highest priority queue; if the packet has no entry in the Elephant Tracker nor in the Flow Manager, we assume it is a mouse and assign it to the medium priority queue; if the packet belongs to a flow predicted as elephant, it is assigned to the lowest priority queue. Our priority queues schema is much less fine-grained than pFabric, where the exact residual flow size is used to determine packet priority. Yet, we argue that residual flow size is hard to obtain and that the proposed coarse-grained priority definition is easier to implement and sufficient to reduce FCT.

We evaluate our approach using YAPS [31], a packet-based network simulator used in [24] to evaluate pHost and pFabric, two state-of-the-art baselines we compare against. We use the same network topology as in pFabric: the fabric interconnects 144 hosts through 9 leaf switches (top-of-rack) connected to 4 spine switches in a full mesh. Each leaf switch has sixteen 10 Gbps downlinks, and four 40 Gbps uplinks. The simulator randomly generates 1M flows based on user-provided CDFs, which we extract from CAIDA, MAWI, and UNI traces. To simulate ML hints (DUMBO or pHeavy), we use the model confusion matrix computed on the respective trace's test minutes. Then we instrument the confusion matrix to simulate authentic predictions, i.e., mimicking the model performance one could expect on the trace. This process allows us to reasonably evaluate ML-based approaches at various loads (i.e., number of concurrent flows), even on the UNI dataset that contains fewer flows than the other traces. Results reported in Figure 7a show that pFabric and pHost perform near-optimally with an average slow down with respect to the ideal one within 1.13-1.28×, and 1.13-1.48× respectively. We remark that these two best-performing policies are often impractical, as both of them assume exact knowledge of the flow size by modifying end-hosts (this often requires changes to all the applications involved in the system). Instead, DUMBO only relies on hints extracted from the first $k = 5$ packets to assign scheduling priorities. Nonetheless, our system often provides performance close to pFabric and pHost (average slow down 1.19-2.64× with respect to the ideal), while operating without any dependence on end hosts and with much fewer priority queues. We note that both ML-enabled systems largely outperform host-agnostic FIFO scheduling. Comparatively, pHeavy scores an average slowdown 1.18-4.22×. We note that DUMBO outperforms pHeavy on all traces but MAWI, where all approaches perform on par. This is due to the flow sizes distribution of MAWI, which features over 90% of flows with less than 5 packets and 99% of flows with less than 16 packets, hence making it a less challenging trace for scheduling. Finally, we remark that DUMBO could manage more priority queues by leveraging the model prediction confidence, i.e., the number of trees votes in agreement. This optimization is impractical for pHeavy because it features a single tree per stage.



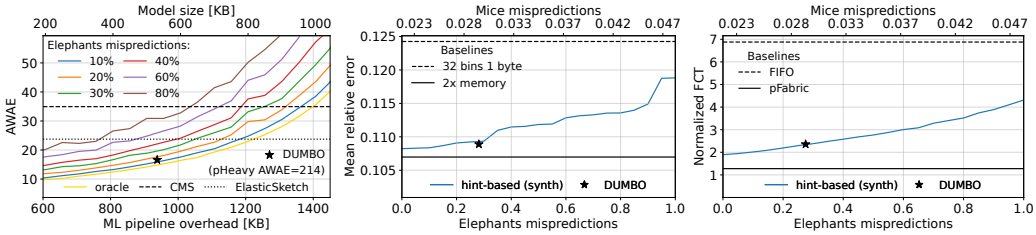
(a) Flow scheduling slowdown. (b) IAT quantile estimation error. (c) Flow size estimation error.

Fig. 7. Benefits of our hint-based monitoring approach over three use cases.

Flow IAT distribution estimation. To evaluate DUMBO performance on the IAT distribution estimation use case, we measure the 75th and 95th IAT quantiles and defer results on other quantiles to Appendix B. We compare against baselines that use the same 33 MB total memory budget to estimate the IAT distribution of all flows. The first baseline allocates one DDSKetch of 32×1-byte buckets for each flow, and the second baseline a DDSketch of 16×2-byte buckets per flow. For DUMBO and the pHeavy-based solution we use DDSketches of two different sizes based on the predicted flow class: 32×2 bytes buckets for elephants (high-accuracy DDSketches), and 31×1 byte buckets for mice (low-accuracy DDSketches). For these two hint-based approaches, we allocate 2^{15} high-accuracy DDSketches and 2^{20} low-accuracy ones, hence using the same amount of memory as for the baselines (33 MB). The rationale behind this strategy is that most flows do not require a high number of buckets for accurately estimating IAT, hence wasting memory with oversized DDSketches. We argue that longer flows i.e., Elephants, naturally lead to a higher number of IAT measurements, thus to a higher chance of overflowing counters.

The custom packet-level simulator we use to evaluate IAT and flow-size distributions [4] is mainly written in Rust, replays the test traffic traces, and simulates the full set of DUMBO system components. When a flow exits the Flow Manager, model predictions are gathered from an ML model written in Python and wrapped to the simulation using ONNX[5]. Results in Figure 7b assess the advantages of hint-based approaches over both baseline configurations, in terms of mean relative estimation error, plotting the median over the 10 test minutes. As a reference, we also report an "ideal" performance when allocating accurate DDSketches for all the flows (32 2-byte buckets — i.e., doubling the memory). Remarkably, DUMBO provides errors close to this ideal setup while halving the memory requirements. Finally, the pHeavy-based solution reports a higher error compared to DUMBO, although it achieves decent end-to-end performance partly because of the heavy-lifting operated by the Flow Manager: the information it stores allows for exact quantile computation for a fraction of the flows with less than $k = 5$ packets. This explains why in some cases DUMBO and pHeavy even outperform the "ideal" baseline.

Flow size estimation. Finally, we evaluate the flow size estimation use case in terms of average weighted absolute estimation error (AWAE) as in [22, 26]. We size the ML-enabled pipelines and the two baselines to fit three different memory budgets, namely 1, 1.5, and 2 MB. The first baseline is a plain Count-Min Sketch (CMS) with 2 rows of 3-byte buckets, and the second is ElasticSketch (ES) [50], a popular sketch for flow size estimation. For DUMBO and the pHeavy-based solution, the Elephant Tracker is augmented with an additional 3-byte counter for each entry, to monitor exact flow sizes for the elephants. The approximate monitoring of short flows is delegated to the Mice Tracker which consists of a CMS of 2 rows, with 3-byte buckets in the first row and 2-byte buckets in the second row. This allows for a wider second row, while the first one serves as a backup in case of overflow and mitigates the effect of model misprediction. Similar to our hint-based approach,



(a) Flow size estimation. (b) IAT estimation, 95th quantile. (c) Flow scheduling.

Fig. 8. Impact of mispredictions on use cases end-to-end performance (CAIDA trace).

ES also dedicates exact counters for monitoring the size of elephants. However, ES does not use a model to provide hints but rather uses a dynamic mechanism called *ostracism* to determine which flows should be removed from the exact counters and moved to the mice CMS instead. For ES, we evaluate different repartitions for the memory allocated between the elephant exact counters and the mice CMS, reporting results with the best memory partitioning. Finally, due to the lack of flows in the UNI traces, we downsize the memory setup by 100× when simulating with this dataset.

As for the previous use case, we use our custom simulator [4] with test traffic traces. Results averaged over the 10 test minutes are shown in Figure 7c. We remark that, for all considered memory budgets, DUMBO outperforms both the plain CMS and state-of-the-art ElasticSketch. On CAIDA, where the number of flows to measure is substantial, the pHeavy-based solution exhibits poor performance at low memory budgets. This is explained by pHeavy misclassifying many mice (low precision) and saturating the Elephant Tracker; this leaves no space for legitimate elephants that would then pollute the mice CMS. The lower the memory budget, the smaller the mice CMS and the greater the impact of low-precision hints. Notably, on MAWI (or UNI), the relative (or absolute) small amount of flows with more than 5 packets leads to lower penalization for mice flows misclassified as elephants by both hint-based approaches. Additionally, pHeavy is impractical to deploy in a low-memory regime because it extracts more features from incoming packets, hence requiring much more memory for the Flow Manager (≈1.6 MB on CAIDA, exceeding the 1.0 MB system-related memory of DUMBO by ≈600 KB that are detracted from the task-related memory).

6.3 Impact of model performance on downstream tasks

In this subsection we analyze DUMBO end-to-end performance with respect to memory overhead and model quality. First, we investigate the impact of model size to identify the maximum memory overhead that can be tolerated. Second, we characterize the impact of model mispredictions on the performance of the downstream tasks. To do so, we use our custom simulator [4] run with CAIDA traces and the system configuration introduced in the previous section (cf. Table 3).

Figure 8a shows the impact of model size on the flow size estimation task when 1 MB of memory is available for the task itself. The figure plots the size of the ML model alone (top axis) and the overhead of the whole ML pipeline (bottom axis). DUMBO is denoted with a black star (542 KB model, 949 KB pipeline, 19 AWAE). The impact on the end-to-end performance also depends on the ML hints quality, here characterized in terms of the percentage of misclassified elephants. Our findings assess that the ML pipeline overhead greatly impacts the deployment feasibility and effectiveness on the downstream task. For instance, a model that wrongly classifies 20% of elephants as mice still brings more than 20% benefits in terms of AWAE compared to ElasticSketch as long as the ML pipeline fits 1 MB. However, for an ML pipeline overhead of 1.2 MB, this benefit disappears as the end-to-end error exceeds ElasticSketch's. Finally, with such hints quality, a pipeline requiring more than 1.35 MB is not beneficial even when compared to a simple CMS.

Moreover, the plot shows that the end-to-end performance deterioration due to pipeline overhead is more severe than the deterioration due to hints degradation. For instance, a pipeline that fits 1 MB is better than ElasticSketch with up to ≈ 40% elephants mispredictions. Last, Figure 8a also reports oracle performance (i.e., a model without any mispredictions). Interestingly, even an oracle would not bring any benefit over the ElasticSketch and CMS baselines if the ML pipeline exceeds ≈1.2 MB and ≈1.4 MB respectively. In our experiments, we notice that, counterintuitively, the mice misprediction rate plays a more impactful role on the end-to-end performance: for a 1 MB pipeline, a model featuring 3.5% (4.5%) mice mispredictions performs worse than the CMS baseline (plots in Appendix B). This result is due to the imbalanced nature of traffic: even a slight increase in mice misprediction rate translates into a huge overall number of mice flows being wrongly classified as elephants, hence preventing legitimate elephants from entering the Elephant Tracker.

Figure 8b analyzes the impact of misprediction rates on the IAT estimation task. As per previous sections, for this use case the memory allocated to the task itself is one order of magnitude larger than the pipeline size: therefore, we do not show how the trade-off changes varying the pipeline size, as effects are negligible. The plot reports mean relative error on the 95-th quantile (other quantiles in Appendix B) varying elephants misprediction rate from 0 to 1 and mice misprediction rate consequently i.e., so that ≈20K flows are classified as elephants. Results show that a model featuring low elephant mispredictions provides end-to-end performance similar to the baseline that uses twice the same amount of memory i.e., allocating high precision DDSketches for all the flows. Notably, for this use case, pHeavy reports good performance, as its memory overhead has relatively less impact on the IAT estimation task. In general, our tests show that up to ≈30% of elephants misprediction i.e., ≈3% of mice mispredictions, a hint-based system provides a relative error only ≈2% higher compared to the ideal configuration that uses twice the memory budget.

Last, Figure 8c analyzes the impact of mispredictions on the scheduling use case, simulating 1M flows with a load of 0.8. Similar to the previous use cases, the plot reports the normalized FCT when varying the elephants misprediction rate between 0 and 1 (and mice misprediction rate accordingly to classify ≈ 20K flows as elephants). We recall that our system considers only three priority queues (one for each predicted class and one for the Flow Manager), in contrast to pFabric which considers each flow's residual size to assign fine-grained priorities. As a consequence, even a perfect binary hint-based system that correctly classifies all elephants cannot reach the performance of pFabric. Yet, DUMBO only increases FCT by 1.85× compared to pFabric (solid black line), without any end-host involvement. Additionally, we note that the model ability to detect elephants (recall) is more important than its precision in this three-queues setting. This explains the FCT gap between pHeavy and DUMBO on CAIDA (+28% recall), which is further visible on UNI (+56% recall).

7 CONCLUSION

In this paper, we presented DUMBO, an end-to-end system design to generate and benefit from approximate flow size predictions in the data plane. These predictions are generated using a lightweight ML model based on custom Random Forests. We investigated the model training and update routine to enable accurate and stable classification of incoming flows into elephants or mice, using both packet-level and flow-level features aggregated from the first k packets. Furthermore, we carefully studied the data plane implementation feasibility of the components at play in an ML-enabled pipeline, analyzing the various trade-offs to consider, and presented an FPGA SmartNIC prototype. Finally, we evaluated the proposed system on three network tasks (flow scheduling, inter-arrival time estimation, and flow size estimation). From this analysis, we emphasize the significance of accurately predicting mice over elephants due to the heavily imbalanced classification problem at hand. Thanks to its good hints quality and careful data plane pipeline design, DUMBO is more efficient compared to both classic and ML-enabled data plane solutions in the state of the art.

REFERENCES

[1] 2006. The MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/.

[2] 2019. The CAIDA Anonymized Internet Traces Dataset. https://www.caida.org/catalog/datasets/passive_dataset/.

[3] 2023. AMD OpenNIC Project. https://github.com/Xilinx/open-nic.

[4] 2024. *DUMBO Simulator*. https://github.com/cpt-harlock/DUMBO

[5] 2024. *Open Neural Network Exchange (ONNX)*. https://github.com/onnx/onnx

[6] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. 2020. Classic Meets Modern: A Pragmatic Learning-Based Congestion Control for the Internet. In *Proceedings of the 2020 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'20)*. 632–647.

[7] Aristide Tanyi-Jong Akem, Beyza Bütün, Michele Gucciardo, and Marco Fiore. 2022. Henna: Hierarchical machine learning inference in programmable switches. In *Proceedings of the 1st International Workshop on Native Network Intelligence*.

[8] Aristide Tanyi-Jong Akem, Michele Gucciardo, and Marco Fiore. 2023. Flowrest: Practical Flow-Level Inference in Programmable Switches with Random Forests. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*.

[9] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 435–446.

[10] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565* (2016).

[11] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*. 3971–3988.

[12] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. 2020. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. USENIX, 667–683.

[13] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2020. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1172–1185.

[14] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. https://pages.cs.wisc.edu/~tbenson/IMC10_Data.html. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 267–280.

[15] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[16] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Martin Leeser, Miriam Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. 2022. The future of FPGA acceleration in datacenters and the cloud. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 3 (2022), 1–42.

[17] Leo Breiman. 2001. Random Forests. *Machine Learning* 45 (2001).

[18] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. 2022. pForest: In-Network Inference with Random Forests. arXiv:1909.05680

[19] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[20] Alexander D'Amour, Katherine Heller, Dan Moldovan, Ben Adlam, Babak Alipanahi, Alex Beutel, Christina Chen, Jonathan Deaton, Jacob Eisenstein, Matthew D Hoffman, et al. 2022. Underspecification presents challenges for credibility in modern machine learning. *The Journal of Machine Learning Research* 23, 1 (2022), 10237–10297.

[21] Nicola Di Cicco, Amir Al Sadi, Chiara Grasselli, Andrea Melis, Gianni Antichi, and Massimo Tornatore. 2023. Poster: Continual Network Learning. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 1096–1098.

[22] Elbert Du, Franklyn Wang, and Michael Mitzenmacher. 2021. Putting the "Learning" into Learning-Augmented Algorithms for Frequency Estimation. In *38th International Conference on Machine Learning*. PMLR.

[23] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. 51–66.

[24] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2019. pHost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. 1–12.

[25] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record* 30, 2 (2001), 58–66.

[26] Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. 2019. Learning-Based Frequency Estimation Algorithms. In *International Conference on Learning Representations*.

[27] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The p4 to netfpga workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 1–9.

[28] Arthur S. Jacobs, Roman Beltiukov, Walter Willinger, Ronaldo A. Ferreira, Arpit Gupta, and Lisandro Z. Granville. 2022. AI/ML and Network Security: The Emperor has no Clothes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*.

[29] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal quantile approximation in streams. In *Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 71–78.

[30] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the 2020 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'20)*. 514–528.

[31] Gautam Kumar, Akshay Narayan, and Peter Gao. 2016. *YAPS Network Simulator*. https://github.com/NetSys/simulator

[32] Chunghan Lee, Yukihiro Nakagawa, Kazuki Hyoudou, Shinji Kobayashi, Osamu Shiraki, and Takeshi Shimizu. 2015. Flow-Aware Congestion Control to Improve Throughput under TCP Incast in Datacenter Networks. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, Vol. 3. 155–162.

[33] Jong-Hyouk Lee and Kamal Singh. 2020. Switchtree: in-network computing and traffic analyses with random forests. *Neural Computing and Applications* (2020), 1–12.

[34] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*. 44–58.

[35] Charles Masson, Jee E Rim, and Homin K Lee. 2019. DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proc. VLDB Endow.* 12, 12 (2019), 2195–2205.

[36] Oliver Michel, Roberto Bifulco, Gabor Retvari, and Stefan Schmid. 2021. The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Computing Surveys (CSUR)* 54, 4 (2021), 1–36.

[37] Michael Mitzenmacher. 2021. Queues with small advice. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM, 1–12.

[38] Andrea Monterubbiano, Raphael Azorin, Gabriele Castellano, Massimo Gallo, Salvatore Pontarelli, and Dario Rossi. 2023. Memory-efficient Random Forests in FPGA SmartNICs. In *Companion of the 19th International Conference on emerging Networking EXperiments and Technologies*. 55–56.

[39] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A centralized" zero-queue" datacenter network. In *Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'14)*. 307–318.

[40] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Bianchi. 2019. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. 531–548.

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

[42] Alessandro Rivitti, Roberto Bifulco, Angelo Tulumello, Marco Bonola, and Salvatore Pontarelli. 2023. eHDL: Turning eBPF/XDP Programs into Hardware Designs for the NIC. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 208–223.

[43] Alessio Sacco, Flavio Esposito, and Guido Marchetto. 2020. A Federated Learning Approach to Routing in Challenged SDN-Enabled Edge Networks. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. 150–154.

[44] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. 2022. Continuous in-network round-trip time monitoring. In *Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'22)*. 473–485.

[45] Burr Settles. 2011. From theories to queries: Active learning in practice. In *Active learning and experimental design workshop in conjunction with AISTATS 2010*. JMLR Workshop and Conference Proceedings, 1–18.

[46] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. 2022. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22)*. 513–533.

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

[48] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. 2019. Is advance knowledge of flow sizes a plausible assumption. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. 565–580.

[49] Zhaoqi Xiong and Noa Zilberman. 2019. Do switches dream of machine learning? Toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets'19)*. 25–33.

[50] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*. 561–575.

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

[52] Zongyi Zhao, Xingang Shi, Zhiliang Wang, Qing Li, Han Zhang, and Xia Yin. 2021. Efficient and Accurate Flow Record Collection With HashFlow. *IEEE Transactions on Parallel and Distributed Systems* 33, 5 (2021), 1069–1083.

[53] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2022. IIsy: Practical in-network classification. *arXiv preprint arXiv:2205.08243* (2022).

[54] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2022. Automating in-network machine learning. *arXiv preprint arXiv:2205.08824* (2022).

[55] Changgang Zheng and Noa Zilberman. 2021. Planter: seeding trees within switches. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. 12–14.

A ADDITIONAL MODEL ANALYSIS

Here we provide additional analysis on the data used for training and evaluation and on the model itself. In particular, in Figure 9 we analyze the flow size distributions of each trace, broken down by protocol. As expected, ICMP traffic features smaller flows. Figure 10 motivates us to exclude ICMP traffic from our analysis, as it would artificially inflate model performance on MAWI. On the other hand, this has almost no impact on CAIDA and UNI traces instead (as they feature almost no ICMP flows). Nevertheless, we report model performance including ICMP traffic in Figure 11, while we base our main evaluation on TCP and UDP traffic in the paper.

Figure 12 shows a simplified explainable tree extracted with the TrusteeML library [28] on CAIDA. Note that this is a heavily-pruned version of the tree, that aims at interpretability at the expense of performance.

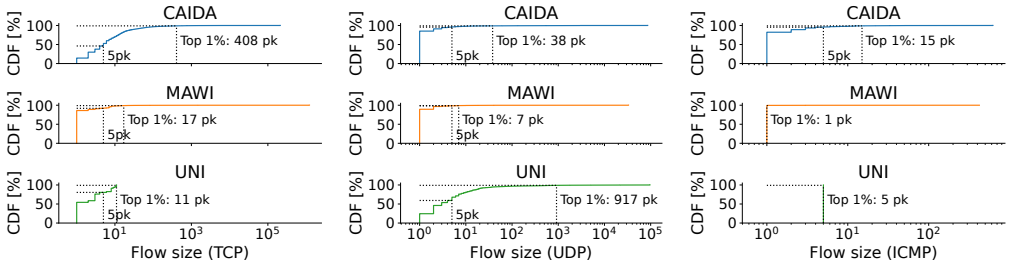


Fig. 9. Flow sizes distributions by protocol for the 50th minute of each trace.

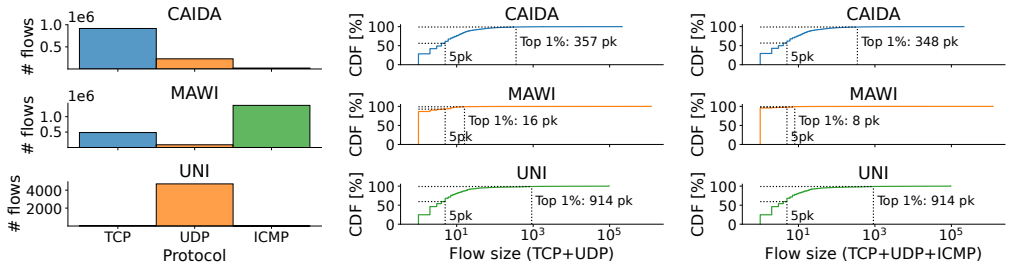


Fig. 10. Traffic analysis for the 50th minute of each trace. UNI features almost no ICMP flows.

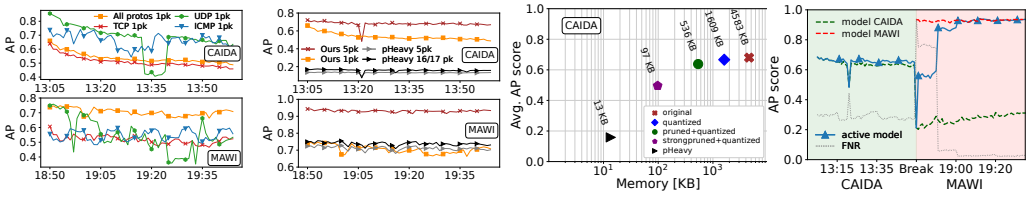


Fig. 11. Model evaluation including all three protocols (i.e., TCP, UDP, and ICMP) on CAIDA and MAWI. The large volume of ICMP flows makes the classification task easier, especially on MAWI.

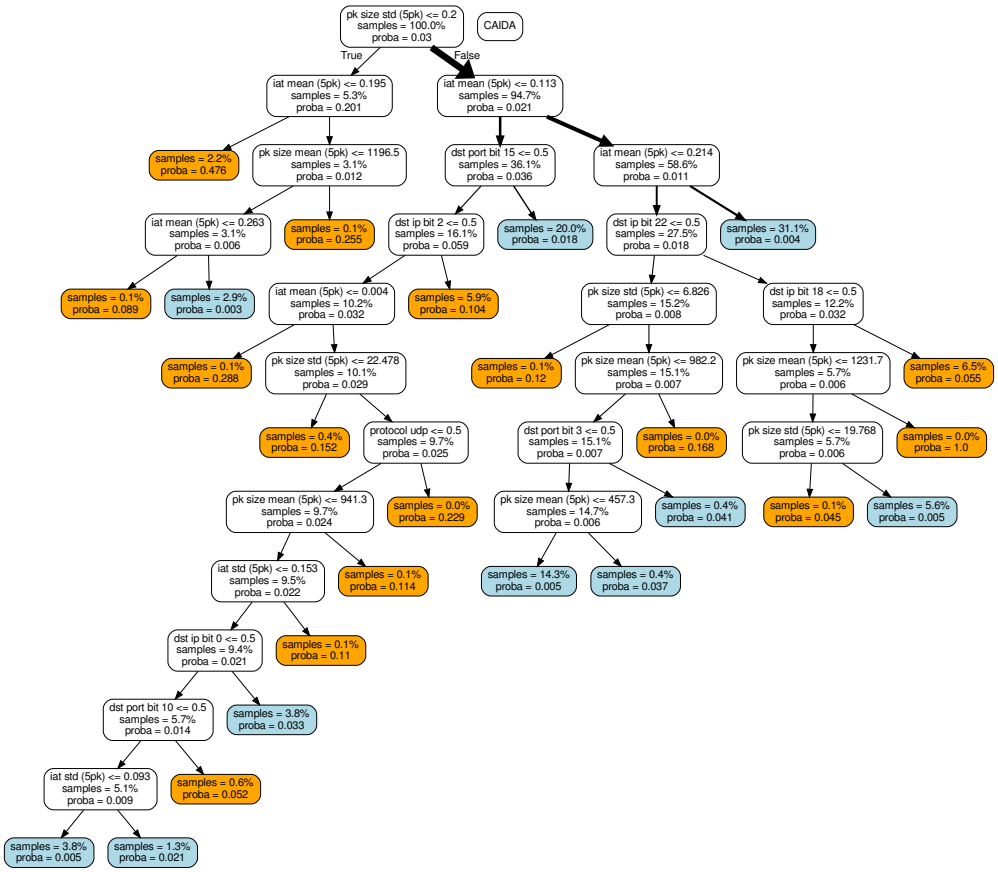
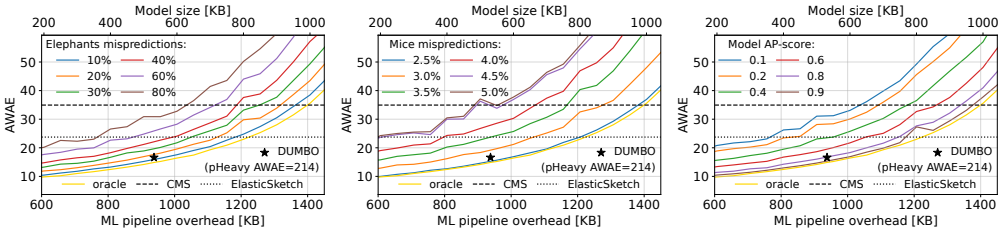


Fig. 12. Simplified view of the explainable tree extracted with TrusteeML [28]. The probability threshold is ≈0.043 for ≈20K elephants. Blue leaves correspond to Mice and orange leaves to Elephants. Samples proportions may not sum to 100% because of top-k branches pruning.

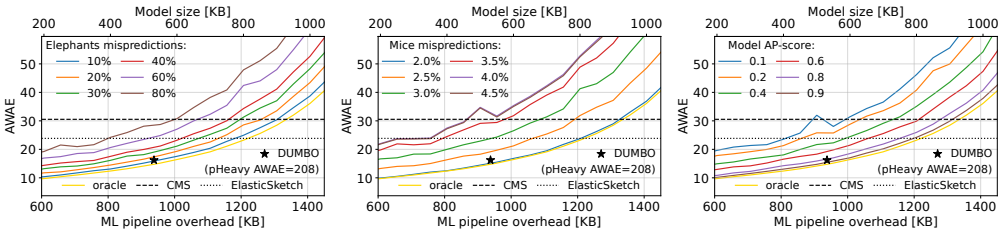
B ADDITIONAL TRADE-OFF ANALYSIS

Here we provide additional results concerning the trade-offs between memory overhead and end-to-end performance, and how these are impacted by the model mispredictions (CAIDA trace). In particular, Figure 13ab show the impact of elephants and mice mispredictions varying memory overhead on combined TCP and UDP traffic (Figure 13a also appears in the main paper), while Figure 14ab show results for TCP-only traffic. Additionally, we show in Figure 13c and Figure 14c how the offline AP-score metric translates into end-to-end average weighted absolute error for combined TCP and UDP traffic, and TCP-only respectively. Last, in Figure 15 and Figure 16 we show how changing the misprediction rates affects the Mean Relative Error (MRE) when computing multiple quantiles for the IAT estimation use case, respectively on TCP+UDP and TCP-only traffic (Figure 15c also appears in the main paper). Note that the DUMBO marker (black star) corresponds to the actual model performance while solid lines refer to simulated performance trade-offs by artificially modifying the model confusion matrix.



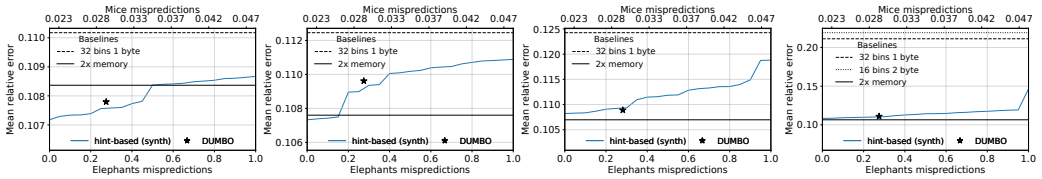
(a) Impact of FNR. (b) Impact of FPR. (c) Impact of AP score.

Fig. 13. Impact of mispredictions on flow size estimation use case (TCP and UDP).



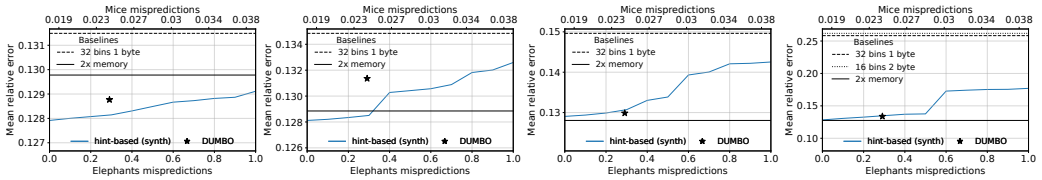
(a) Impact of FNR. (b) Impact of FPR. (c) Impact of AP score.

Fig. 14. Impact of misprediction on FSE use case (TCP only).



(a) 75-th quantile. (b) 90-th quantile. (c) 95-th quantile. (d) 99-th quantile.

Fig. 15. Impact of misprediction on IAT use case (TCP and UDP).



(a) 75-th quantile. (b) 90-th quantile. (c) 95-th quantile. (d) 99-th quantile.

Fig. 16. Impact of misprediction on IAT use case (TCP only).

Received June 2023; revised December 2023; accepted January 2024