FENXI: Fast In-Network Analytics

Massimo Gallo
Huawei Technologies Co. Ltd
France
massimo.gallo@huawei.com

Alessandro Finamore
Huawei Technologies Co. Ltd
France
alessandro.finamore@huawei.com

Gwendal Simon
Huawei Technologies Co. Ltd
France
gwendal.simon@huawei.com

Dario Rossi
Huawei Technologies Co. Ltd
France
dario.rossi@huawei.com

ABSTRACT

Live traffic analysis at the first aggregation point in the ISP network enables the implementation of complex traffic engineering policies but is limited by the scarce processing capabilities, especially for Deep Learning (DL) based analytics. The introduction of specialized hardware accelerators i.e., Tensor Processing Unit (TPU), offers the opportunity to enhance processing capabilities of network devices at the edge. Yet, to date, no packet processing pipeline is capable of offering DL-based analysis capabilities in the data-plane, without interfering with network operations.

In this paper, we present FENXI, a system to run complex analytics by leveraging TPU. The design of FENXI decouples forwarding operations and traffic analytics which operates at different granularities i.e., packet and flow levels. We conceive two independent modules that asynchronously communicate to exchange network data and analytics results, and design data structures to extract flow level statistics without impacting per-packet processing. We prototype FENXI on a general-purpose server and evaluate its performance in both adversarial and realistic network conditions. Our evaluation shows that FENXI is able to offer DL processing to 100 Gbps linecards with a limited number of resources, while also dynamically adapting to network conditions.

CCS CONCEPTS

• **Hardware → Analysis and design of emerging devices and systems**; • **Networks → In-network processing**.

KEYWORDS

Traffic Measurement, Deep Learning, Real-Time

ACM Reference Format:
Massimo Gallo, Alessandro Finamore, Gwendal Simon, and Dario Rossi. 2021. FENXI: Fast In-Network Analytics. In *Proceedings of ACM/IEEE Symposium on Edge Computing 2021 (SEC '21).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the last decade, Deep Learning (DL) has become a fundamental analytics technique in some fields of computer science such as computer vision and natural language processing. DL-based analytics is rapidly gaining momentum in the network community too, with the promise of enabling complex Artificial Intelligence (AI)-based traffic engineering. Proposals that use DL model to improve traffic engineering include application identification [10, 32], analytics that enables traffic differentiation [26, 46], malware and attack detection [19, 27] used by firewall applications, and anomaly detection [41] by troubleshooting tools. However executing pre-trained DL models (a.k.a. inference) is still a complex operation, which requires significant processing capabilities that are not commonly found in the network edges. Indeed, the research community has concentrated efforts in designing systems that delegate inference to external devices, typically hosted in a cloud environment [16, 28].

We consider traffic monitoring operated at the network edge, typically in the first aggregation point after the so-called "last-mile" in broadband networks. Despite the increasing interest in both industry and research communities, the adoption of DL has yet to transform network management and traffic monitoring in this scenario. On the one hand, current approaches which offload DL inference to the cloud, alleviate the demand for increasing physical resources at the edge [11]. On the other hand, cloud offloading is both ineffective for latency-sensitive use-cases (e.g., when decisions should be taken within a few Round-Trip Time (RTT) of the flow life cycle) and unable to deal with the constant increase of broadband link capacity (e.g., resulting in overwhelming control traffic to the cloud services) as pointed out in [45].

In this paper we study the design of an inference system that leverages low cost and low power consumption *hardware accelerators* [34] such as edge Tensor Processing Unit (TPU) [3, 4], which will be included in the next-generation network cards [7]. We design, implement, and extensively

benchmark Fast In-Network Analytics (FENXI), a system capable of DL inference at high speed. With FENXI, we revamp the packet processing pipeline from the Network Interface Card (NIC) to the storage of the inference result, contributing to the field in two ways: we report a deep dive into a challenging system with multiple contrasting objectives, and we propose practical solutions to every element of the system.

Contribution #1: By implementing and testing FENXI under realistic and extreme conditions, we reveal the complexity of DL analytics in edge network scenarios.

- We benchmark DL hardware accelerators: a Graphic Processing Unit (GPU), a TPU, and a multi-core CPU. We used a state-of-the-art DL model designed for traffic analytics [10] to test the performance of the accelerator. The benchmark report in Section 2 highlights the trade-off between inference speed (to sustain throughput), delay (to meet application requirements), and energy consumption.
- We examine the characteristics of real network traffic to identify system requirements. Our analysis emphasizes two requirements: (i) high-throughput operations considering the number of packets per second (since information is extracted from individual packets) and the number of flows per second (since analytics apply on flows). (ii) low-delay operation since the delay in inferring the analytics is not only a matter of processing a data stream but also an application requirement that comes from the traffic characteristics and the analytics usage, e.g., post-mortem analytics are not useful for traffic management.

Contribution #2: We introduce FENXI architecture, together with a thorough analysis of the potential bottlenecks, exposing pitfalls that should be avoided in the design.

- We present in Section 4 the architecture of the Flow manager, which is responsible for *extracting features* from the flows of packets without interfering with the traffic forwarding. We corroborate the design choices with a set of micro-benchmark to illustrate the performance of each individual building block.
- We present in Section 5.1 a *dynamic batching system*, which addresses the threefold requirements of sustaining throughput, maintaining low delay, and reducing energy usage. The concept of grouping multiple data in batches is fundamental in inference serving systems but introduces latency. Some researchers have regarded batching as a not viable solution for latency-sensitive applications [24, 36]. We study this supposed mismatch and improve over state-of-the-art solutions [23] by introducing a mechanism to reduce the waste processing usage and hence of energy.
- We introduce in Section 5.2 a dedicated *caching system*, which we specifically design for analytics of network flows. Previous papers have shown that caching for packet inference suffers from header entropy [40]: we address this problem by designing approximate caching policies that suit packet series in flows and evaluate them through a set of micro-benchmark.
- We finally evaluate in Section 6 the performance of the whole FENXI system under scenarios that are the most challenging with respect to our objective of implementing analytics in the data path.

2 CONTEXT AND REQUIREMENTS

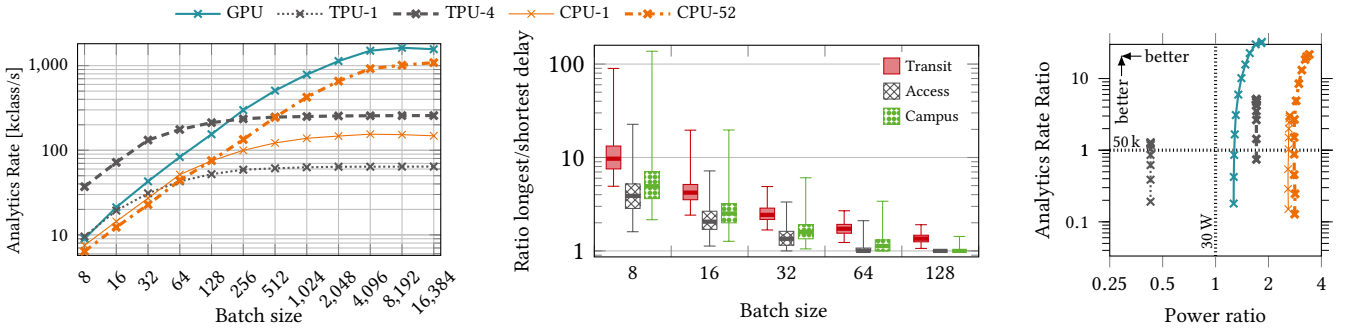
We describe in this section the parameters and constraints that interplay in the design of traffic analytics pipelines. Whereas the main principles of FENXI broadly apply to multiple DL analytics models and network scenarios, we further introduce a specific case study with the aim of clarifying the challenges and providing tangible numerical objectives. We first describe the regarded case study, and then, we emphasize the main operational points of the system.

2.1 Case Study

Instead of an in-breadth analysis of multiple use-cases, we opt for an in-depth analysis of one use-case. We focus on *application identification* as a classic example of flow-level traffic analytics. The identification of the specific application related to a flow is a strategic network management operation. For this task, the inference is triggered after having observed a sufficient (but small) number of packets for each flow [13, 17]. Traffic classification has received growing attention from researchers in the DL community [10]. We chose traffic classification among other analytics due to its challenging requirements: (i) all first packets of every flow should be stored, (ii) it applies on the majority of flows, and (iii) application requires fast classification. In comparison, other traffic analytics have less stringent requirements.

We implement for this paper a 1D-Convolutional Neural Network (CNN) model, which size (about 100 k weights) is smaller than typical 2D CNN models used for image processing, but is significantly larger than the toy-case models used in the related system work [39, 40]. We disregard details about model training and accuracy in this paper, but we point out that the model is equivalent to the one used in [12] trained with over 200 applications labels, which is about ten (four) times the typical (maximum) number of classes considered in the literature [10].

Flow classification requires the extraction of IP packets for the analysis of network flows, as identified by the 5-tuple at the network layer (IP addresses and ports of both source and destination plus protocol). For this specific analytics, FENXI triggers, at the K^{th} packet of each flow, the DL inference based on the size of packets. For other analytics, FENXI is capable of using other flow information and triggers. By default in this paper, we use K equal to 10, which corresponds



(a) Classification rate vs. batch size.

(b) Distribution of per-second ratio of longest/shortest batch completion delays.

(c) Analytics vs. power relative to two targets: 50 kclass/s and 30 W.

Figure 1: Throughput requirements, boxplots show 99^{th}, 75^{th}, 25^{th}, and 1^{st} percentiles.

to a trade-off between the amount of information to analyze and the delay to process the analytics [10]. Formally we extract from each flow a *series* S containing K information, which are extracted from the K first packets $s_1, \ldots, s_K$ of the flow. We denote by $\mathcal{S}_K$ the set of series of size K. In the case of traffic classification, the information s_i extracted from the i^{th} packet of a flow consists in packet length and direction. Other analytics may require information such as Inter-Arrival Time (IAT) [10] or transport-level flags.

We benchmark the performance of the DL model on the hardware accelerators. We ran preliminary experiments on servers equipped with Intel Xeon Platinum 8164 CPUs @ 2.00GHz (L1/L2/L3 caches 32 data+32 instruction)/1024/36608 kB) and 100 Gbps Mellanox MCX515A-CCAT ConnectX-5. As DL offload hardware, we used either a Huawei Atlas 300I:3010 Inference Card (equipped with 4× Ascend 310 chips), or an Nvidia V100 GPU. To provide a fair comparison between TPU and GPU, we did not port the model to the native Huawei Mind Studio stack; we rather cross-compiled the original TensorFlow model for the Huawei Atlas engine.

FENXI targets network devices that operate at the edge of the Internet and contribute to network management operations. To analyze the requirements of such devices, we studied two representative datasets: (*i*) a gateway that connects a *Campus* network to the Wide Area Network (WAN) Internet, this device is representative of enterprise private networks; and (*ii*) a Point-of-Presence (POP) router that connects a residential *Access* network to the Internet. Since both datasets are private, for the sake of transparency, we also used a third dataset (called *transit*), which comes from the public MAWI project.[1] For each dataset, we extracted relevant traffic characteristics by identifying all flows by means of the classic IP 5-tuple, from which we extracted packet time-stamps and lengths that are reported in Table 1 which is used to motivate system requirements.

[1] http://mawi.wide.ad.jp/ – extracted 2020-02-12, 2020-03-04, 2020-03-25, 2020-04-08, 2020-05-27, 2020-06-03, 2020-06-10

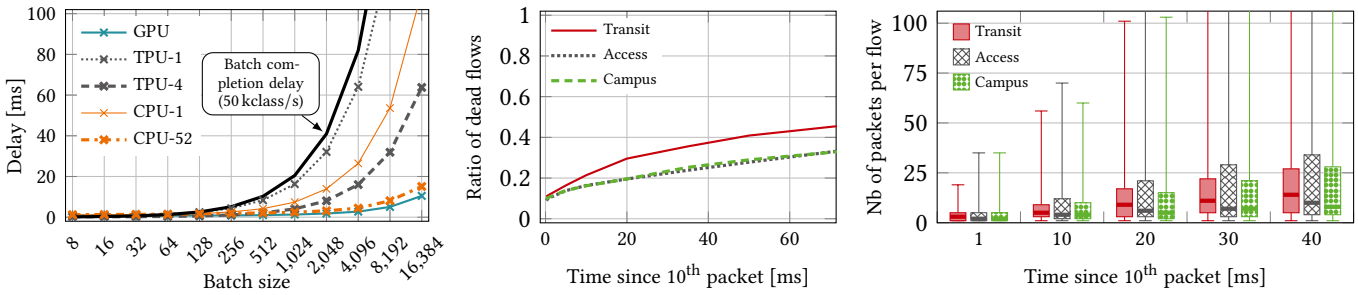
	Traffic characteristics				Rate at 100 Gbps		
dataset	**vol.** [GB]	**#pkts** [M]	**#flows** [k]	**#series** [k]	**packet** [Mpps]	**flow** [kflows/s]	**series** [kclass/s]
access	765	858	3963	2481	14.0	64.7	40.5
transit	870	923	2476	1968	13.2	35.5	28.3
campus	483	516	2700	1718	13.3	69.8	44.4
average	706	765	3046	2055	13.4	56.6	37.3

Table 1: Traffic characteristics in real datasets.

2.2 Throughput

The capacity of a network device to sustain a given throughput is an essential feature, not only from the traditional perspective of data rate (measured in bit per second) but also from the perspective of flow analytics (measured in classification per second). FENXI design should aim to not sacrifice the former to perform the latter. We also have to consider the foreseen growth of both data rates (due to higher throughput in local networks) and flow analytics (due to the increase of connected devices). While today's edge routers typically deal with throughput in the order of single-digit Gigabits per second (Gbps), the requirements for the next-generation routers consider up to 100 Gbps. In the following, to put FENXI into stress, we will use this configuration for 100 Gbps network routers as a reference.

We show in Table 1 the total number of packets, flows and series (i.e., flows with more than ten packets) in our datasets. We use these traffic characteristics to derive the required performance in terms of packet (i.e., Mpps) and series processing speed (i.e., class./s) for the reference 100 Gbps at maximum



(a) Batch processing and completion delay vs. batch size.

(b) Ratio of dead short-lived flows after the 10^{th} packet.

(c) Number of received packets for long-lived flows after the 10^{th} packet.

Figure 2: Delay requirements. Boxplots show 99^{th}, 75^{th}, 25^{th}, and 1^{st} percentiles.

load. We conclude that FENXI deployed in a single 100 Gbps linecard should be able to sustain 15 Mpps and 50 kclass/s.

We then evaluate whether the hardware accelerator can sustain such an analytics rate. We report analytics throughput in Figure 1a in terms of classification per second achieved by CPU (both 1 and 52 cores), GPU (640 cores available), and TPU (both 1 and 4 cores). The main software parameter that impacts the analytics rate is the size of the *batch*, i.e., the number of series that are grouped for parallel processing. To evaluate this aspect, we performed a 60 s long stress test by submitting a stream of inference tasks of a given batch size (i.e., a new submission is triggered without waiting for the previous to complete), while measuring the time between each submission and the related output delivery. We show in Figure 1a that, with a batch size greater than 64, every hardware accelerator can sustain a target 50 kclass/s. Interestingly, whereas CPU and GPU are optimized for large batch size, the 4 TPU chips combined are significantly faster in processing small batches than both 52 CPUs and GPU.

The stress test however does not correctly represent real traffic behavior. Network traffic is subject to *instantaneous* high-load over a short period, i.e., bursts [23], and this can be a challenge regarding the overall system throughput. To estimate whether incoming data bursts also reflect in the series arrival rate, we measured during 10 min of each dataset the *batch completion delay*, which is the time needed to complete a batch of a given size. Thus, for a batch $B = 8$ it is the time needed for 8 consecutive new series to enter in the system. We computed the ratio between the longest and the shortest completion delays observed within each second. A high ratio indicates that the batch completion delays are subject to a high variation. Results in Figure 1b show that batch completion delays vary significantly, especially with small batches. Hence, we highlight that the results in Figure 1a on the maximum classification rate for a given configuration addresses one aspect of the analytics rate while FENXI should also be able to absorb bursts, by dynamically adapting the batch sizes to sudden series arrivals.

Finally, we analyze the throughput with respect to energy consumption, which is a key performance criterion in today's edge devices. Figure 1c illustrates the tradeoff of analytics throughput versus power usage for varying batch sizes as a scatter plot of the power (x-axis) and classification rate (y-axis) normalized with respect to reference values of 30 W that is the power consumption of an idle GPU and 50 kclass/s. The top-left square highlights desirable configurations although the bottom-right one includes operational regions that should be avoided. For instance, only one hardware configuration (TPU with 1 core) with a couple of settings can meet the requirement of our configuration (30 W and 50 kclass/s), while the other hardware configurations fail in at least one dimension.

2.3 Delay

The second essential feature of a traffic analytics pipeline is the delay between the time at which the analytics can be processed (in our case study, it is the time at which the K^{th} packet arrives at the device) and the time at which the result of the analytics can be exploited.

Simplifying, such delay is split into two components: the *analytics processing delay* and the *batch completion delay*. We report in Figure 2a the analytics processing delay for multiple batch sizes. We emphasize here the trade-off between throughput and delay since the results from Figure 1a call for the use of a larger batch, but the processing of a large batch generates a significant delay. Furthermore, we added in Figure 2a a line to represent the batch completion delay for a 50 kclass/s constant series arrival rate. The batch completion delay is dominant in comparison to the analytics processing delay. We conclude that setting the batch size is key: a large batch results in a long delay, while a small one leads the inference system to run in a sub-optimal operational regime.

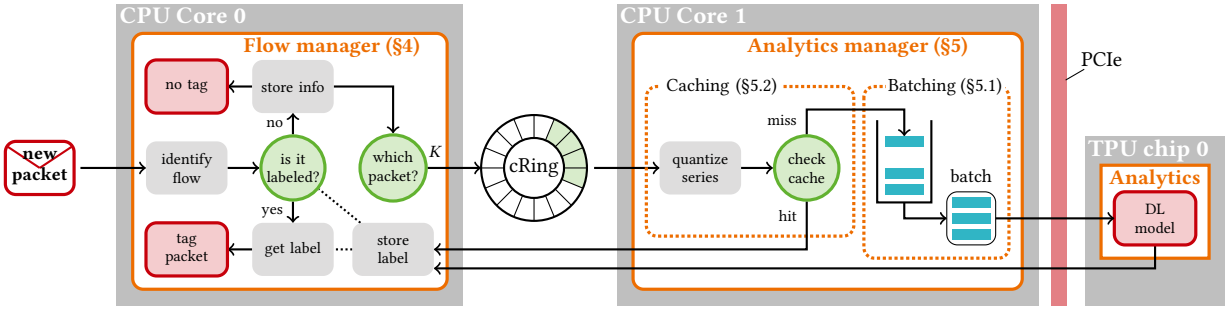


Figure 3: Modular view of the FENXI system.

We highlight now the impact of delay in flow analytics when regarded from the application of the classification result. We distinguish flows in *short-lived* and *long-lived*. The former represents the vast majority of the flows while the latter is responsible for the vast majority of data volume. In our datasets, considering the target of analytics after the 10^{th} packet we identify a flow as short-lived if it has less than 35 packets, long-lived otherwise. FENXI targets to identify the application label for a short-lived flow before observing the flow last packet to avoid *post-mortem* analytics. Indeed, such a label is meant to tag flow's packets before forwarding them. However, the life expectation of short-lived flows after the 10^{th} packet arrival ranges from a few milliseconds to minutes (see Figure 2b). Long-lived flows on the contrary are likely classified before their end, but FENXI targets early classification to enable effective traffic differentiation [26], so the sooner the classification, the better.

Figure 2c presents the number of packets received by long-lived flows since the 10^{th} packet. Waiting for the label drives the system to an increasing number of untagged packets hence deteriorating the performance of the algorithm exploiting such information. We conclude that delay requirement cannot be strictly and universally set, since it also depends on the location of the device in the network (see the differences between datasets). Yet, from our analysis, a target around 10 ms enables a low fraction of both untagged packets and post-mortem analytics.

2.4 Takeaway

We showed in this section that the design of FENXI should take into account multiple, sometimes conflicting, objectives related to throughput, delay, and energy consumption.

We are positive in the capabilities of hardware accelerators to handle the classification rates for regular edge routers, even at 100 Gbps, but we noticed that tailored policies regarding batch size have to be designed to deal with bursts. Whereas it can meet the throughput requirements, the CPU does not offer a viable hardware platform for fast analytics since the energy consumption is too large while it does not offer processing gains. TPU offers a more energy-efficient alternative, especially with regards to its higher processing performance for small batches, while GPU is the option to adopt for large throughput requirements.

The study of IAT in flows revealed that we cannot express any unique delay objective, contrarily to previous work on inference service for image processing applications [16]. We recommend small batch sizes to meet a delay objective around 10 ms, but we also showed that small batch sizes cannot sustain high throughput nor low power consumption. These three objectives (throughput, delay, and energy) define the operational points of the FENXI system.

3 FINX OVERVIEW

This section introduces FENXI design, which is presented in Figure 3. The FENXI architecture splits the processing into two parts: *Flow manager* and *Analytics manager*, which can hence be deployed on dedicated independent processing units. This design choice is driven by the fact that the two managers operate at different granularities. The former handles every packet entering the system and splits them according to the flow while the latter is executed on group of series, i.e., features collected from flows with more than K packets.

A Flow and an Analytics manager together constitute a single processing pipeline and communicate through a communication ring *cRing*, which enables zero-copy and lockless data exchange. In the following, we introduce FENXI building blocks that we will detail in the rest of the paper.

Flow Manager. The flow manager is FENXI interface towards the system NICs. Received packets are processed according to the steps pictured on the left side of Figure 3. FENXI identifies the flow through the IP 5-tuple. If the flow has been already classified in the past, it forwards the packet with a tag indicating the classification result. If the flow has not been classified yet, FENXI updates its internal data structures and forwards the packet with an empty tag. Furthermore, if the received packet completes a series i.e., this

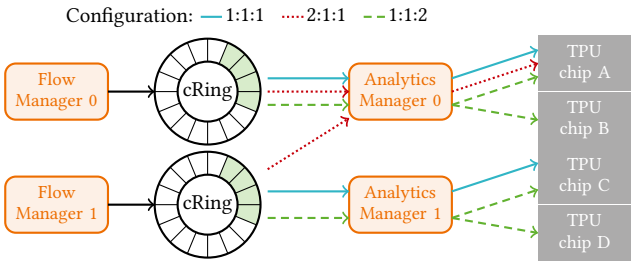


Figure 4: Pipeline deployment strategies.

packet is the K^{th} packet of this flow, FENXI forwards the extracted features and the flow tuple to the next element in the pipeline (the analytics manager) via the cRing.

To fulfill device requirements (cf. Section 2), the flow manager must reduce per-packet processing overhead, such that any function carefully considers hardware and software capabilities. As stated in Section 7, previous work on *offloaded* inferring services have either not addressed line rate preprocessing (for example in the literature on image processing [25, 47]) or considered less demanding offloading tasks (for example load-balancing [23]). Feature extraction at line rate is critical for in-device analytics implementation. We describe the feature extraction process in Section 4.

Analytics Manager. The Analytics manager constantly polls the cRing, waiting for flow tuple and series to be pulled. To increase efficiency, software operations of the Analytics manager are tightly coupled to the underlying DL hardware accelerator. To fulfill constraints and requirements expressed in Section 2, we further split the Analytics manager into two sub-modules *Batching* and *Caching* with the twofold objective of improving data transmission and analytics processing speed. Batching is responsible for building groups of series for which analytics will be computed in parallel. Caching acts as a filter before batch composition and stores recently executed analytics to speed up processing.

Each series is processed by the Analytics manager according to the steps pictured on the right side of Figure 3. If the series is cached, the result is immediately returned to the Flow manager. Otherwise a batch of series is formed and, when ready, sent to the Analytics device for processing. In turn, when the result of the batch is ready, the analytics manager forwards the labels back to the Flow manager.

Pipeline deployment strategies. We showed in Section 2 that different networks present different packets, flows, and series rates. Hence, the two pipeline stages can be more or less time and resource-demanding, and bottlenecks do not necessarily happen at the same stage of the pipeline. As such, we design FENXI to have a configurable multi-pipeline system. To reduce system overhead, we only consider lock-free strategies for which the same flow/series information is always handled by the same couple of Flow and Analytics managers, which run on separate processing units.

Thanks to its flexible design FENXI pipeline can be configured to address the processing bottleneck by means of deployment strategies. Figure 4 presents three lock-free multi-pipeline deployment strategies. (*i*) A pipeline in a 1:1:1 configuration is the default configuration with a Flow manager, an Analytics manager, and an analytics processor i.e., the TPU that runs in isolation. (*ii*) The 1:1:2 configuration addresses a bottleneck in the analytics processing throughput. In this configuration, a single Analytics manager balances the analytics load across more than one TPU chips. (*iii*) The 2:1:1 configuration copes with scenarios with high packet arrival rates but mid/low series arrival rates. In such a case a single Analytics manager interacting with a single TPU can retrieve series from multiple Flow managers.

Finally, for each strategy, FENXI can scale up by instantiating multiple pipelines in parallel by exploiting modern NICs dynamic forwarding of incoming traffic into different receive queues by using Receive Side Scaling (RSS) to load balance packets acquisition across different Flow manages.

According to our analysis in Section 2, a single TPU can sustain the load for 100 Gbps. Nonetheless, we deploy a system with the multi-pipeline 1:1:1 configuration, which maintains a lower load on the hardware accelerator and can absorb instantaneous high series arrival rate over short periods.

4 FLOW MANAGER

The Flow manager module aims to extract the features that are relevant for the analytics. A high-level diagram of its operations is presented in Figure 3. Packets are split into flows according to the classic IP 5-tuple and then processed depending on three cases. (*i*) The packet is part of a *labeled* flow, i.e., FENXI previously ran analytics on this flow, which resulted in a label. In this case, the forwarding plane performs actions on the packet with respect to the label, e.g., *tagging* the packet before forwarding it. It is important to highlight that when the packet is part of an unlabeled flow, rather than waiting for the label to be computed, FENXI forwards the packet unmodified. We say that the packet is *untagged*. In the latter scenario, we further distinguish two cases: (*ii*) if the packet is not the K^{th} packet of the flow, FENXI only updates the flow state. (*iii*) If the packet is the K^{th} packet of the flow, the analytics processing for this flow can be triggered. FENXI forwards the *features* extracted from the first K packets (e.g., a series of packet properties), and passes it to the next pipeline element, which asynchronously retrieves it via the lock-less communication ring, cRing.

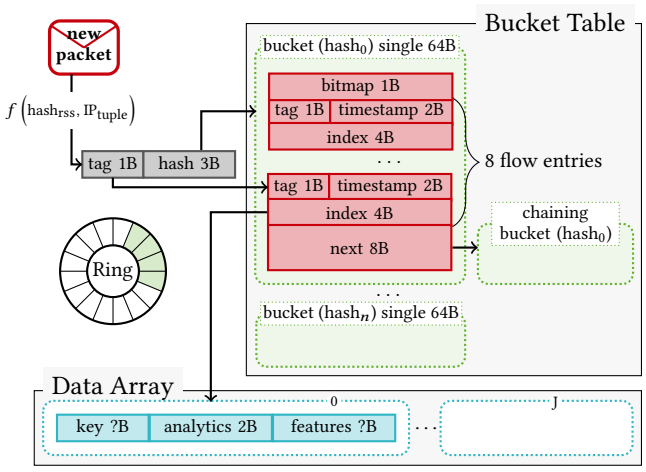


Figure 5: Flow manager Hash table.

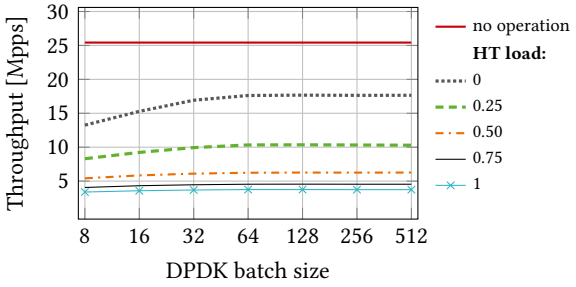


Figure 6: Single worker flow manager processing speed with increasing batch size and hash table load.

4.1 System Design

The Flow manager data plane is currently designed around Data Plane Development Kit (DPDK) but, in principle, it can be ported to similar packet processing frameworks for general-purpose servers [35] or smart NICs [1, 6].

At startup time, the Flow manager instantiates several workers, one for each pipeline, which independently handles ongoing flows dispatched by the linecard with RSS. To keep track of ongoing flows, FENXI uses a hash table (see Figure 5) with multiple entries buckets (i.e., 8) coupled with a data array, which is addressed using array indexes retrieved through a dedicated ring buffer. The data array size (i.e., the maximum number of entries that can be stored by the flow manager) and the number of buckets can differ and hence used to artificially control the number of collisions within a single bucket at the expense of memory efficiency.

When a packet arrives, the hash table bucket position is computed using the three least significant bytes of a *hash* value computed on the IP 5-tuple. To reduce as much as possible the packet processing time the Flow manager relies on the fact that modern linecards and NICs compute symmetric hash values [44] (i.e., the same hash is computed for both directions of a flow) on the IP 5-tuple to load balance the flows across the different receive queues i.e., RSS, and attach such value to packet's metadata. To mitigate the unbalance issues reported by Woo and Park [44] when using the Teoplitz symmetric hash function we compute our hash as:

$$f(hash_{rss}, IP_{tuple}) = hash_{rss} \oplus IP_{src} \oplus IP_{dst}$$

where $\oplus$ represents the bitwise XOR operation, which provides a balanced hash function at a very low processing cost.

The hash table bucket is stored in a single cache line (i.e., 64 Bytes) and composed by a *bitmap* used to quickly check the occupied entries in the bucket, eight flow entries, and a *next* value, which points to an external, dynamically allocated, memory area to handle chaining. Each flow entry stores a 1 Byte *tag* extracted from the hash value, a 2 Bytes *timestamp*, and a 4 Bytes *index*. We use the hash tag, which corresponds to the first byte of the hash value itself, for a quick comparison to accelerate lookups in case of a high number of collisions [14]. The coarse grain timestamp (in seconds) stores the last time the flow entry was accessed (e.g., for packet labeling or for statistics update) to lazily remove inactive flow entries based on the *stale* timeout system parameter. Finally, we use the *index* to point to the element of the data array that stores the information of the flow. A separate ring buffer is used to keep track of data array indexes, which are available for storing flow information.

The data array stores (i) the flow's *key* (i.e., the IPv4/IPv6 5-tuple used during the lookup), (ii) the result from the analytics (i.e., the *label* stored as an atomic variable to avoid contention), (iii) the features extracted from the flow (i.e., packet size and direction of the first K packets), and (iv) some statistics about the flow (e.g., the number of packets).

4.2 Micro-benchmark

We ran two micro-benchmarks to better understand FENXI Flow manager performance in isolation (without the Analytics manager), and in worst-case conditions, 64 Bytes packets, using two servers with DPDK 20.02 (see the technical description in Section 2), which are directly connected via a 100 Gbps link.

In the first test, we evaluated the processing speed of the Flow manager with a single worker for increasing DPDK batch size i.e., the maximum number of packets the system will receive in parallel. The data array size is fixed to 2^{19} (524 k) with 2^{17} (131 k) buckets. At startup, we preloaded the hash table to reach a fixed load factor ($load = elements/size(data\_array)$). During a 30 s time frame, we

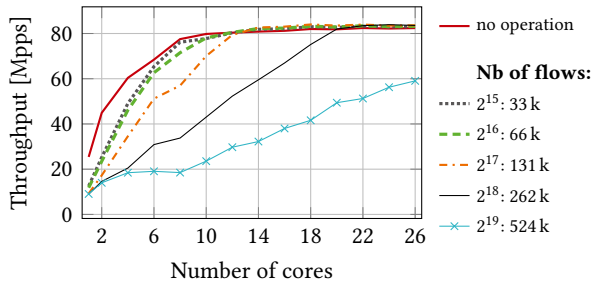


Figure 7: Flow manager scalability at hash table load 0.5 with increasing number of workers and flows.

sent 64 Bytes packets belonging to the pre-loaded flows at maximum speed to the Flow manager from the second server running Moongen [20]. Figure 6 shows the maximum number of packets retrieved in parallel by the DPDK framework for different hash table loads (at load 0, a single flow is used during the test). The *no operation* reference corresponds to Flow manager workers that receive and forward packets without performing any additional operation. Large batch sizes improve the packet processing efficiency, leading to a higher speed, especially for lower loads, which are not far from the limit highlighted by the *no operation* reference line. Moreover, higher hash table loads lead to slower processing speed, which highlights the fact that a hash table set with a wrong dimension can become a bottleneck.

In the second test, we evaluated the processing speed with an increasing number of Flow manager workers (one per core) and a different number of flows. We pre-loaded the hash table such that it reached a 0.5 load factor and sent 64 Bytes packets belonging to such flows. We present in Figure 7 the processing speed for a bucket size that corresponds to a quarter of the size of the hash table. The more workers are used by the flow manager, the higher is the processing speed. Moreover, processing speed is limited at 80 Mpps due to hardware limitations (the theoretical limit is 144.88 Mpps) imposed by the PCI Express 3.0 x16, which our 100 Gbps NIC is attached to. Similar limitations are reported by Neugebauer et al. [31]. We also observe that the processing speed does not only depend on the hash table load factor but also on the number of concurrent flows. Hence, the hash table timeout for removing old elements needs to be correctly set to avoid too many stale flows, e.g., 30 s.

Our micro-benchmarks and the preliminary system requirements analysis in Section 2 allow us to conclude that the Flow manager can sustain 100 Gbps traffic by using only two or three workers (i.e., cores) depending on the traffic characteristics and hash table load.

Algorithm 1: Timeout based dynamic batching

```
At timeout T expiration
begin
    r ← length of cRing;
    B ← min {b ∈ ℬ : b ≥ r};
    batch[0..r] ← cRing[0..r];
    padding ← B − r;
    if padding then
        batch[r..B] ← pad[0..padding];
    end
    Process(batch);
end
```

5 ANALYTICS MANAGER

The analytics manager module takes as input series composed of flows' features and flow tuple extracted by the Flow manager and perform the desired analytics i.e., early flow classification. Similarly to the flow manager, at startup, the analytics manager instantiates several workers, one per pipeline according to the multi-pipeline strategy. Each worker retrieves series ready to be analyzed by constantly polling the dedicated cRing (cf. Figure 4). The analytics manager leverages two key components, namely (*i*) *Dynamic Batching* and (*ii*) *Approximate Caching*. In the remainder of this section, we further detail the design and internal architecture of batching and caching processing.

5.1 Batching

Grouping multiple series into one batch to offload the analytics presents two main advantages: it reduces the communication overhead by amortizing transmission overhead and enables a faster computation through parallel processing and memory access. According to our preliminary benchmark in Section 2, the bigger the batch, the shorter the per-series processing delay i.e., the time needed for sending, processing, and receiving the analytics results of a single series. Hence, if we only take into account processing throughput, the bigger the batch, the better.

However, big batches take longer to be filled, which can be a problem at low series arrival rates. System designers have thus to find a tradeoff between analytics throughput and operational delay with respect to the series arrival rate and the application requirement. Overall, adopting a static batch size leads to systems working well only on a single operational point. Conversely, we implement in FENXI a dynamic batching strategy with variable batch size to tradeoff analytics throughput, delay, and processing power.

The concept of *dynamic batch* consists in running the DL model with variable batch size [16]. Due to the limited amount of available resources in DL accelerators, a DL model cannot be used with any batch size; it is defined with one

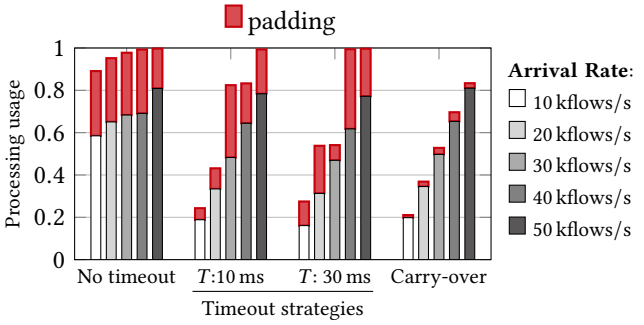


Figure 8: Processing usage for TPU (1 core). The Carry-over strategy is set with timeout at 10 ms and $\Phi = 0.2$.

unique batch size B. Dynamic batch is then implemented by hosting multiple models, each one with a different batch size as in [5, 8]. A naive approach to dynamic batching is presented in Algorithm 1. Let $\mathcal{B}$ be the set of batch sizes of the implemented DL models. When the system decides to fire analytics processing (e.g., using a timeout), r series are waiting in the cRing. The Analytics manager creates a new batch from these r series by selecting the model with batch size B such that B is the smallest batch size greater than r in $\mathcal{B}$. If the batch is not complete, the system may add $B - r$ padding series before sending it to the hardware accelerator.

To better understand the behavior of such an algorithm in realistic network conditions, we simulated system behaviors by extracting single-chip TPU's performance profile reported in Section 2 for batch sizes equal to 2^x with $x \in [3..10]$. The simulation input is then generated by a Poisson process that inputs a flow randomly chosen from flows extracted from the *campus*, *home*, and *transit* datasets at a given average series arrival rate λ. Figure 8 presents the total hardware accelerator usage in the simulated scenario, which is further split in padding and series with different timeout values i.e., left, no timeout, and center fixed timeout. Notice that we here consider timeouts that are in the order of magnitude of (i) the RTT to get analytics as soon as possible and (ii) the analytics processing delay reported in Section 2.

When no timeout applies, the system sends a new batch immediately when the accelerator is ready. In this case, the processing usage is close to 1 even when the series arrival rate is low. This strategy is inefficient in two aspects: (i) at low arrival rates, the number of series r is low, so the system should add a large number of padding, which is a waste of resources; (ii) since the number of series r is low, the size of the batch B is also low, which is less efficient in terms of processing throughput. The timeout addresses the problem of over-utilization of the accelerator since the hardware is used only a fraction of time at low series arrival rates. We

Algorithm 2: Carry-over dynamic batching

```
At timeout T expiration
begin
    r ← length of cRing;
    B ← min {b ∈ ℬ : b ≥ r};
    padding ← B − r;
    if padding/B > Φ then
        B ← max {b ∈ ℬ : b ≤ r};
        batch[0..B] ← cRing[0..B];
    else
        batch[0..r] ← cRing[0..r];
        if padding then
            batch[r..B] ← pad[0..padding ];
        end
    end
    Process(batch);
end
```

observe however that a large amount of computational power is spent on processing padding, regardless of the timeout.

To prevent the accelerator to uselessly process padding, we design a custom dynamic batching system. FENXI adopts both concepts of dynamic batching and timeout T because they provide an efficient basis for trade-off throughput and application delay. Note that since the cRing is pulled in pool mode a real timeout does not exist, we nonetheless present the batching strategy with a real timeout for ease of presentation. We augment the timeout batching strategy with a *Carry-over* system to control the padding as illustrated in Algorithm 2. Let Φ be a threshold in $[0, 0.5]$ that sets the maximum fraction of padding accepted in the batch. When the timeout is reached, the system contains r series. Let B be the smallest batch size in $\mathcal{B}$ greater than r. If the padding necessary to complete the batch is greater than Φ, the system scales back to a smaller batch size B', which is the largest batch size in $\mathcal{B}$ smaller than r. This way, we do not include in the batch the latest series that arrived in the series ring. These series wait for the next batch that will be processed.

In essence, the Carry-over policy (i) adds some extra delay for a subset of series, but these series have just arrived in the system so the extra delay compared to the timeout is small; and (ii) enables control of the fraction of padding in the system. The rightmost part of Figure 8 presents the results in the simulated scenario for the Carry-over policy with $\Phi = 0.2$ and timeout = 10 ms demonstrating that it is able to control both padding and processing efficiency with respect to no timeout and fixed timeout batching policies. We further analyze the performance of our batching strategy in Section 6 using the FENXI prototype.

Algorithm 3: Prefix cache

```
At timeout T expiration
begin
    for z ← 0 to len(cRing) do
        S_K ← cRing [z];
        S_δ ← s_{1≤i≤δ} ∈ S_K;
        if l_{S_K} ← Cache.lookup(S_δ) then
            pull(cRing[z]);
        end
    end
    batchComposition();
end
At analytics result reception
begin
    for z ← 0 to B do
        S_K ← batch[z];
        S_δ ← s_{1≤i≤δ} ∈ S_K;
        Cache.insert(S_δ);
    end
end
```

5.2 Caching

A trained DL model is deterministic. In the case of traffic classification it can be abstracted as a non-linear function $f(\cdot)$ that maps a series $S \in \mathcal{S}_K$, i.e., with features extracted from the first K packets, to a class, i.e., label $l \in \mathcal{L}$. The label set $\mathcal{L}$ depends on the analytics. In the traffic classification example, we can classify 200 different applications. The function f is not injective: multiple series can have the same label $l \in \mathcal{L}$. Based on this observation and on the fact that multiple flows have the same series (see Table 1), we foresee the benefits of implementing a cache, which stores popular analytics computation, to both speed up analytics and reduce the load on the hardware accelerator.

In our context, the caching system C stores C entries in the form of keys and values, where the key is a series $S_K \in \mathcal{S}_K$ and the value is an associated label l_{S_K}. An incoming flow for which the extracted series S_K is cached in C is directly classified with the stored label. The performance of such cache, in terms of hit ratio increases when (i) the number of different flows having the same series is large and (ii) the distribution of flows per series (popularity) is skewed.

To increase the hit-ratio of the cache, a cache designer can implement *approximate caching* (a.k.a. similarity caching) [21, 30, 33] by (i) reducing the set of keys $\mathcal{P}$ in the cache and (ii) applying an approximate function that maps the set of input series $\mathcal{S}_K$ to a smaller set of keys $\mathcal{P}'$. Without loss of generality, we focus in the context of our use case on *prefix caching*, where any cached key is a subset of the series S_K, i.e., the key set $\mathcal{P}'$ is the subset $\mathcal{S}_\delta$ where $\delta < K$. Formally, we define a function $q_\delta(\cdot)$ that transforms a series in $S_K \in \mathcal{S}_K$ to a series $S_\delta \in \mathcal{S}_\delta$ where, for any $1 \leq i \leq \delta$, and

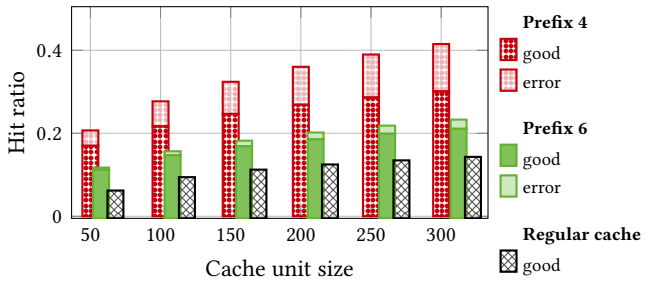
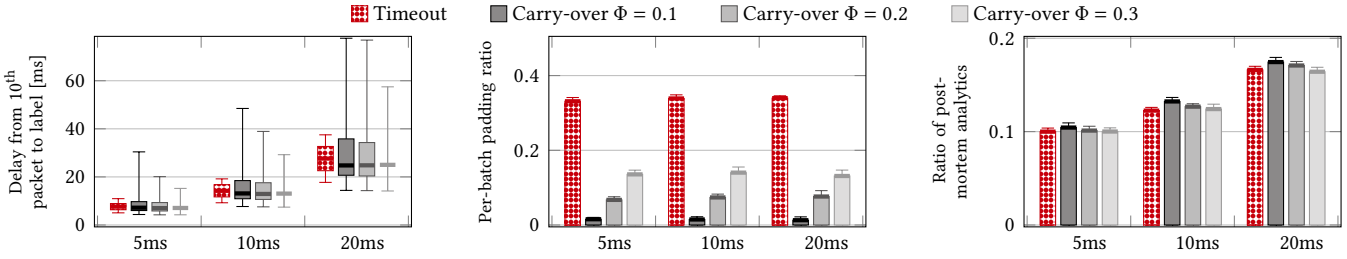


Figure 9: Prefix cache hit-ratio for prefixes $\delta = 4$ and $\delta = 6$, in comparison to regular cache with 10 packets.

for any series' feature $s' \in S_K$ with $s' = q_\delta(s) \in S_\delta$, $s'_i = s_i$. Notice that other time-series based analytics (e.g., forecast of load or other signals) would equally benefit from policies with very similar implementation (e.g., *postfix caching*, to give more importance to most recent samples), so while the quantitative evaluation is limited to the traffic classification use-case, the qualitative lessons holds to a larger extent.

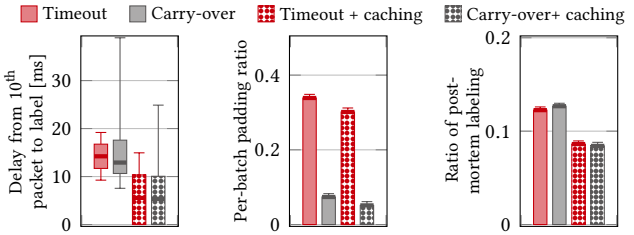
FENXI implement the *prefix cache* as a Least Recently Used (LRU) cache that act as filter before the batch composition described in Algorithm 2. Note that in the real implementation caching and batching modules are entangled, but we present them separately for the sake of clarity. The process at timeout expiration and at analytics result reception is described in Algorithm 3. When the timeout expires, before the batch composition and for each series in the cRing, the approximation function $q_\delta(\cdot)$ is applied to series S_K to obtain the prefix series S_δ for which a cache lookup is performed. In case S_δ has a matching label l, i.e., *Hit*, FENXI tags the series S_K with the label l and pulls the element from the cRing since the analytics is considered as already executed. Notice that, according to the cache replacement policy, *Cache.lookup* also adjusts internal cache data structures e.g., updates the list of least recently used elements for LRU. In case S_δ does not have a matching label l, i.e., *Miss*, the processing continues. Finally, a batch is composed using Algorithm 2. At analytics results reception, FENXI inserts the label l in the cache for the corresponding prefix series S_δ derived from S_K.

For a given hit-ratio target, the prefix cache is smaller than the regular cache. It is however an approximate cache in the sense that it is not guaranteed that the series S_K and S_δ have the same analytics result. In case of a hit, two cases can be distinguished: either (i) the prefix hit does not introduce any additional error, since the label that the DL model would predict l_{S_K} is the same as the one that is stored in the cache for the prefix series l_{S_δ} (that we denote as *Good*); or, (ii) the approximate hit introduces an additional error (that sums up to the DL model error) as the label that the DL model would



(a) Delay to obtain the label. **(b) Ratio of padding.** **(c) Ratio of post-mortem.**

Figure 10: Timeout and Carry-over batching policy. Boxplots show 99^{th}, 75^{th}, 25^{th}, and 1^{st} percentiles of measures while barcharts report median and 99^{th} percentile.



(a) Delay to label **(b) Ratio of padding** **(c) Post-mortem ratio**

Figure 11: Caching impact ($\Phi = 0.2, timeout = 10ms$). Boxplots show 99^{th}, 75^{th}, 25^{th}, and 1^{st} percentiles; barcharts report median and 99^{th} percentile.

predict l_{S_K} is not the same as the one that is stored in the cache for the prefix series l_{S_δ} (that we denote as *Error*).

To evaluate the performance of the caching system in realistic network scenarios we simulated a workload by using a dataset containing 8946 k flows extracted from both *campus* and *access* ones for which we have corresponding labels i.e., the ground truth used to train the DL model. Figure 9 presents hit ratio for different caching strategies where regular cache means that the key used for the cache is S_K with $K = 10$ and prefix 4 and 6 means that the key used for caching analytics results is S_δ with $\delta = 4$ and 6 respectively. The gain brought by the short prefix $\delta = 4$ with respect to regular cache is significant with an increase of hit-ratio by 3× though it also starts affecting the precision of the analytics. More conservative settings such as $\delta = 6$ allows a reduction of the DL workload by approximately 30 %, limiting the error rate to less than 1 %. We further analyze the performance of caching in Section 6 with the FENXI prototype.

6 SYSTEM EVALUATION

Scenario. We evaluated the overall performance of FENXI as a flow classifier that receives packets in the input port and outputs them in the opposite direction with a tag indicating the class (we use the IP field Type of Service) when the analytics is available. Our evaluation ran on two directly connected servers. The characteristics (cf. Section 2) are: each server is equipped with Intel Xeon Platinum 8164 CPUs @ 2.00GHz (L1/L2/L3 caches 32 data+32 instruction)/1024/36608 kB), a 100 Gbps Mellanox MCX515A-CCAT ConnectX-5 NIC and a Huawei Atlas 300I:3010 Inference Card. In the test, one server ran MoonGen [20] while the other one ran FENXI prototype with data array size and buckets equal to 2^{22} (4 M), and stale timeout 30 s. Note that although servers running FENXI are a high-end server we only use a limited amount of resources e.g., 2/4 cores, 1/2 TPU chips, and a few MB of DRAM that can be found (or installed) at the network access. The workload is two-minute-long traces replayed by MoonGen. Since the throughput of our original traces is too low, we generated realistic traces as follows: the *flow* arrival process follows a Poisson process with average λ, while the *packets* conform to the real flow characteristics (i.e., inter-packet delay, size, and the number of packets per flow) randomly chosen from a catalog of 1 M flows extracted from the Access dataset.

Dynamic batching. We started our evaluation by a scenario in which a single 1:1:1 FENXI pipeline (i.e., a flow manager, an ascend manager, and a TPU) was given 50 kflows/s. Figure 10 compare timeout and Carry-over batching policies in terms of, from left to right, time to perform the analytics since 10^{th} packet , padding ratio, and ratio of flows that did not get analytics before their end i.e., post-mortem. The lower the timeout, the smaller the time to get the label. However, we point out that a smaller timeout drives the system to an inefficient operational point, especially for low arrival rate. On the contrary, a higher timeout leads to a higher percentage of post-mortem analytics. Finally, as pointed out in Section 5.1, the Carry-over batching policy helps to control the level of padding trading off additional delay for the time to get the label in some cases i.e., 99^{th} and 75^{th} percentiles. The lower

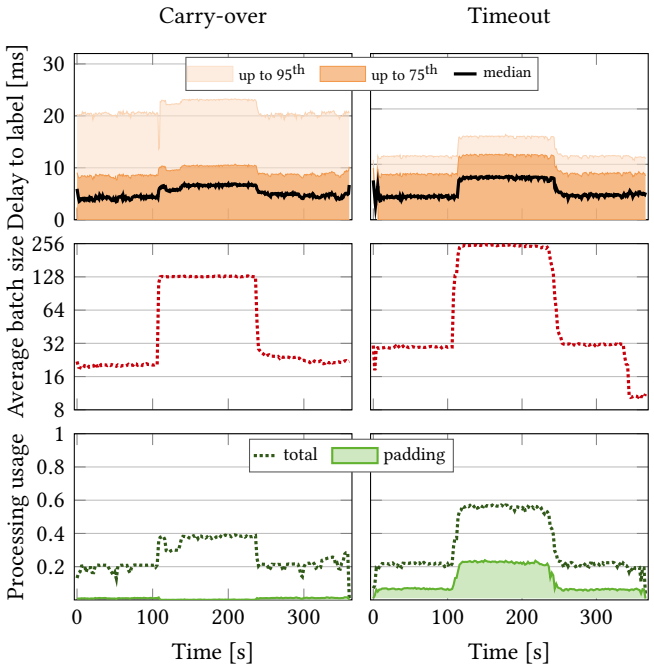


Figure 12: Fixed and Carry-over batching strategies over time under dynamic traffic conditions.

the Φ, the lower the padding and waste of processing and the higher the additional delay.

Approximate caching. In the same scenario we also evaluated the impact of the caching module in FENXI configured with timeout 10 ms and $\Phi = 0.2$ in case of Carry-over. For this scenario, we dimensioned the approximate LRU cache with $\delta = 6$ to have a hit ratio of 0.3 and we do not report here the difference between regular and approximate prefix cache as they would achieve similar results but with the advantage of a smaller memory footprint for the latter, at the price of a higher error. Figure 11 reports the results we obtained with timeout and Carry-over batching policies with or without caching in terms of time to get the label, padding and post-mortem analytics. The results shows that, for both batching strategies, the time to get the label decreases as some of the series i.e., the one for which analytics is cached, are retrieved much faster. Notice also that, for the same reason the variability of the delay increases as well. At the same time, given the fact that the delay to get the label decreases, also the post-mortem analytics ratio decreases.

Flash crowd. Finally we evaluated FENXI performance in a dynamic flash-crowd scenario in which two 1:1:1 pipelines are subject to an input traffic at 10 kflows/s for the first 120 seconds, then 70 kflows/s in the next 120 seconds, and 10 kflows/s for the last part of the experiment. FENXI was configured with a prefix $\delta = 6$ LRU cache and size for which the hit ratio is approximately 0.3. The Carry-over strategy is set with 10 ms timeout and $\Phi = 0.2$. Figure 12 presents the time to get the label, the ratio of TPU usage, and the average batch size over time. Both timeout and Carry-over batching strategies are able to scale up batch size and consequently the processing usage when the traffic load suddenly increases. Notice that, despite the higher processing, the time to get the label only slightly increases. Finally, we highlight the fact that Carry-over is successful in dynamically adapting to the right batch size, reduces energy consumption by lower processing power, and further avoid wasting processing power by greatly limiting the processing power spent for padded input analysis.

7 RELATED WORK

The success of DL technology has ignited interest for its in-network use, so that valuable work started tackling the issue of DL analytics offloading from a networking perspective.

Server-side offloading: The Machine Learning (ML) community has extensively studied the performance of hardware accelerators [18, 34] and the design of offloading mechanisms for data inference to external hardware. However, the existing *inference serving* systems [2] and literature [16, 25, 43, 47] generally target the case of offloading image recognition to GPU-equipped servers in a datacenter. The closest work in this space is *Clipper* [16], whose design also leverages caching and batching, as rather typical weapons in the design space. This solution and subsequent work [43] target cloud-based inference with desirable latency targets (around 20 ms). Recent work [42] also studies the cooperation among edge and cloud in the context of video streaming analytics. However, as pointed out in [45], the volume of data and processing time for DL inference models on network traffic is radically different from that of computer vision applications, so that the system requirement, the processed inputs, and the usage of the analytics output are radically different as well.

Network-wide offloading: Complementary to our work, programmable network devices have been used to assist the processing of heavy workloads [29, 36, 37] some of which have a specific focus on network-assisted DL offloading [36]. For instance, BananaSplit [36] focuses on breaking down DL computation of complex image recognition models across multiple network devices using SmartNIC whereas in our work we adopt the opposite viewpoint and target in-device execution of DL models applied to data plane traffic.

In-device machine learning (ML) offloading: Work that focuses on the programmable data-plane for switches based on Protocol Independent Switch Architecture (PISA) [22, 36,

38, 45] tries to cope with the limited resources that are available for the implementation of ML models. For example [45] adapt several ML models to the match-action table model in P4, while [15] implementing a random forest model to classify traffic. In both cases, the benefits of running the analytics in the data-plane are limited since no immediate packet forwarding decision derives from the analytics. Moreover, the lack of computing and memory resources prevents the implementation of DL models altogether.

In-device deep learning (DL) offloading: Offloading DL inference to hardware accelerators brings better accuracy, the ability to run several DL models in parallel, and to leverage specialized DL frameworks such as TensorFlow [9]. Closest to our work are thus two recent (not peer-reviewed) papers, which explore in-device DL offloading. ASIC is used in [40] for DL inference at packet level but only on toy-models with 3 layers and 21 neurons, i.e., 5000× smaller than the model we use. Smart NIC is used in [39], that however limits model size to 50 *binary* neurons, i.e., 2000× fewer weights, each with a resolution 32× smaller than in our case study. To attain sub-microsecond latency, [39, 40] restrict themselves to such tiny models that it becomes questionable if their execution can have any practical use given the significant distance of such shallow models from the depth needed to embrace the expected benefits of DL. Our work takes the opposite viewpoint and tackles a timely and efficient execution of relevant DL models for edge intelligence by using the appropriate offloading hardware, i.e., TPUs.

8 CONCLUSION

FENXI is the first system to integrate forwarding and advanced analytics capabilities, exploiting TPU hardware acceleration to offer efficient execution of Deep Learning analytics in network devices' data path. Its system design leverages asynchronous communication between forwarding and analytics engines, optimizing the usage of the hardware accelerator by introducing novel dynamic batching and smart caching policies. This makes FENXI capable of high-speed (100Gbps), low-delay (below 10ms), and low-power consumption (on the order of few tens of Watts for the TPU) on off-the-shelf hardware, ultimately paving the way to the deployment of embedded intelligence at the network edge.

REFERENCES

[1] 2018. Huawei IN Series Intelligent NICs. https://e.huawei.com/en/products/servers/pcie-ssd/in-card.

[2] 2020. Amazon Elastic Inference. https://docs.aws.amazon.com/elastic-inference/.

[3] 2020. Ascend 310 chip. https://e.huawei.com/se/products/cloud-computing-dc/atlas/ascend-310.

[4] 2020. Google Coral. https://coral.ai/products/.

[5] 2020. Huawei MindStudio. https://www.huaweicloud.com/intl/en-us/ascend/doc/mindstudio.

[6] 2020. Mellanox BlueField. https://www.mellanox.com/products/smartnic.

[7] 2020. Nividia EGX A100. https://www.nvidia.com/en-us/data-center/products/egx-a100/.

[8] 2020. NVidia Triton Inference Server. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/.

[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *USENIX Symp. on Op. Sys. Design and Impl. OSDI*. 265–283.

[10] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, and Antonio Pescapè. 2019. Mobile Encrypted Traffic Classification Using Deep Learning: Experimental Evaluation, Lessons Learned, and Challenges. *IEEE Trans. Network and Service Management* 16, 2 (2019), 445–458.

[11] Samuel Albanie. 2019. Memory consumption and FLOP count estimates for Convolutional Neural Networks. https://github.com/albanie/convnet-burden.

[12] Cedric Beliard, Alessandro Finamore, and Dario Rossi. 2020. Opening the Deep Pandora Box: Explainable Traffic Classification.

[13] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kavé Salamatian. 2006. Traffic Classification on the Fly. *ACM SIGCOMM Comput. Commun. Rev.* 36, 2 (2006), 23–26.

[14] Fan Bin, G. Andersen David, and Kaminsky Michael. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of USENIX NSDI*. 371–384.

[15] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. 2019. pForest: In-Network Inference with Random Forests. *CoRR* (2019). arXiv:1909.05680 http://arxiv.org/abs/1909.05680

[16] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *USENIX NSDI*.

[17] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. 2007. Traffic classification through simple statistical fingerprinting. *ACM SIGCOMM Comput. Commun. Rev.* 37, 1 (2007), 5–16.

[18] W. Dai and D. Berleant. 2019. Benchmarking Contemporary Deep Learning Hardware and Frameworks: A Survey of Qualitative Metrics. In *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*. 148–155.

[19] Omid E. David and Nathan S. Netanyahu. 2015. DeepSign: Deep learning for automatic malware signature generation and classification. In *Int. Joint Conf. on Neural Networks (IJCNN)*. 1–8.

[20] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *ACM Internet Measurement Conf. (IMC)*.

[21] Fabrizio Falchi, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fausto Rabitti. 2012. Similarity caching in large-scale image retrieval. *Inf. Process. Manag.* 48, 5 (2012), 803–818.

[22] Roberto Gonzalez, Filipe Manco, Alberto García-Durán, Jose Mendes, Felipe Huici, Saverio Niccolini, and Mathias Niepert. 2017. Net2Vec: Deep Learning for the Network. In *Proc. of Big-DAMA Workshop at ACM Sigcomm*.

[23] Panpan Jin, Jian Guo, Yikai Xiao, Rong Shi, Yipei Niu, Fangming Liu, Chen Qian, and Yang Wang. 2019. PostMan: Rapidly Mitigating Bursty Traffic by Offloading Packet Processing. In *USENIX ATC*. 849–862.

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben

Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proc. of the 44th Symp. on Computer Architecture, ISCA*. 1–12.

[25] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *USENIX OSDI*. 611–626.

[26] Fangfan Li, Arian Akhavan Niaki, David R. Choffnes, Phillipa Gill, and Alan Mislove. 2019. A large-scale analysis of deployed traffic differentiation practices. In *Proc. of ACM SIGCOMM*. 130–144.

[27] Gonzalo Marín, Pedro Casas, and Germán Capdehourat. 2018. DeepSec meets RawPower - Deep Learning for Detection of Network Attacks Using Raw Representations. *SIGMETRICS Perform. Evaluation Rev.* 46, 3 (2018), 147–150.

[28] Albert Mestres, Alberto Rodríguez-Natal, Josep Carner, Pere Barlet-Ros, Eduard Alarcón, Marc Solé, Victor Muntés-Mulero, David Meyer, Sharon Barkai, Mike J. Hibbett, Giovani Estrada, Khaldun Maruf, Florin Coras, Vina Ermagan, Hugo Latapie, Chris Cassar, John Evans, Fabio Maino, Jean C. Walrand, and Albert Cabellos. 2017. Knowledge-Defined Networking. *Computer Communication Review* 47, 3 (2017), 2–10.

[29] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of ACM SIGCOMM*. 15–28.

[30] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie D. Enright Jerger. 2015. Doppelgänger: a cache for approximate computing. In *Proc. of the 48th Int. Symp. on Microarchitecture (MICRO)*. 50–61.

[31] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proc. of ACM SIGCOMM*. 327–341.

[32] Fannia Pacheco, Ernesto Exposito, Mathieu Gineste, Cédric Baudoin, and José Aguilar. 2019. Towards the Deployment of Machine Learning Solutions in Network Traffic Classification: A Systematic Survey. *IEEE Commun. Surv. Tutorials* 21, 2 (2019), 1988–2014.

[33] Sandeep Pandey, Andrei Z. Broder, Flavio Chierichetti, Vanja Josifovski, Ravi Kumar, and Sergei Vassilvitskii. 2009. Nearest-neighbor caching for content-match applications. In *Proc. of WWW Conf.* 441–450.

[34] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2019. Survey and Benchmarking of Machine Learning Accelerators. In *IEEE High Perf. Extreme Comp. (HPEC)*.

[35] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *In Proc of USENIX ATC*. 101–112.

[36] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. 2018. Can the Network be the AI Accelerator?. In *NetCompute Workshop at ACM SIGCOMM*.

[37] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *ACM HotNet Workshop*. 150–156.

[38] Giuseppe Siracusano and Roberto Bifulco. 2018. In-network Neural Networks. *CoRR* abs/1801.05731 (2018). arXiv:1801.05731 http://arxiv.org/abs/1801.05731

[39] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. 2020. Running Neural Networks on the NIC. *CoRR* abs/2009.02353 (2020). arXiv:2009.02353 https://arxiv.org/abs/2009.02353

[40] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, and Kunle Olukotun. 2020. Taurus: An Intelligent Data Plane. arXiv:cs.NI/2002.08987

[41] T Van, Hai Anh Tran, Sami Souihi, and Mellouk Abdelhamid. [n.d.]. Network troubleshooting: Survey, Taxonomy and Challenges.. In *Proc. of International Conference on Smart Communications in Network Technologies (SaCoNeT)*.

[42] Nigade Vinod, Lin Wang, and Henri Bal. 2020. Clownfish: Edge and Cloud Symbiosis for Video Stream Analytics. In *ACM/IEEE Symposium on Edge Computing (SEC)*.

[43] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. 2018. Rafiki: Machine Learning as an Analytics Service System. *Proc. VLDB Endow.* 12, 2 (2018), 128–140.

[44] Shinae Woo and KyoungSoo Park. 2012. Scalable TCP Session Monitoring with Symmetric Receive-side Scaling. http://an.kaist.ac.kr/~shinae/paper/2012-srss.pdf.

[45] Zhaoqi Xiong and Noa Zilberman. 2019. Do Switches Dream of Machine Learning?: Toward In-Network Classification. In *18th ACM Workshop HotNets*.

[46] Zhiyuan Xu, Jian Tang, Jingsong Meng, Weiyi Zhang, Yanzhi Wang, Chi Harold Liu, and Dejun Yang. 2018. Experience-driven Networking: A Deep Reinforcement Learning based Approach. In *IEEE Infocom*. 1871–1879.

[47] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *USENIX ATC*. 1049–1062.