FloWatcher-DPDK: lightweight line-rate flow-level monitoring in software

Tianzhu Zhang, *Member, IEEE,* Leonardo Linguaglossa, *Member, IEEE,* Massimo Gallo, Paolo Giaccone, *Senior Member, IEEE,* and Dario Rossi, *Senior Member, IEEE*

***Abstract*—In the last few years, several software-based solutions have been proved to be very efficient for high-speed packet processing, traffic generation and monitoring, and can be considered valid alternatives to expensive and non-flexible hardware-based solutions. In our work, we first benchmark heterogeneous design choices for software-based packet monitoring systems in terms of achievable performance and required resources (i.e., the number of CPU cores). Building on this extensive analysis we design FloWatcher-DPDK, a DPDK-based high-speed software traffic monitor we provide to the community as an open source project. In a nutshell, FloWatcher-DPDK provides tunable fine-grained statistics at packet and flow levels. Experimental results demonstrate that FloWatcher-DPDK sustains per-flow statistics with 5-nines precision at high-speed (e.g., 14.88 Mpps) using a limited amount of resources. Finally, we showcase the usage of FloWatcher-DPDK by configuring it to analyze the performance of two open source prototypes for stateful flow-level end-host and in-network packet processing.**

***Index Terms*—Network traffic monitoring, high-speed packet processing, per-flow packet measurement, Intel DPDK.**

I. Introduction

EVALUATING the performance of experimental devices and network applications requires intensive measurement campaigns on real prototypes, where multiple variables come into play. The procedure follows the guidelines indicated by RFC 2544 on benchmarking network devices, as illustrated in Fig. 1. In case of open-loop experiments, a traffic generator (TX) transmits packets to the Device Under Test (DUT) at a given rate/pattern, typically the worst-case for stress test scenario (e.g., a stream of 14.88 Mpps 64B-packets on 10 Gbps Network Interface Controllers). To measure its performance (maximum sustainable throughput, packet loss, or latency, etc.), the DUT in turn relies the received packets to a traffic monitor (RX). In closed-loop experiments, on the other hand, the traffic monitor captures packets exchanged between transmission and reception ends of the DUTs (i.e., a client-server application) and evaluates the performance.

The TX component presents two design approaches. The first one is leveraging expensive commercial equipments (i.e., hardware traffic generators), capable of generating line-rate traffic with high precision yet providing low-to-none programmability. The second approach is resorting to software traffic generators, which might generate less accurate traffic, but provides a higher degree of flexibility at a much lower cost. A similar discussion also applies for the RX component: hardware solutions can provide accurate measurements on a specific set of preset variables. However, monitoring non-default variables may be challenging if not impossible. In contrast, software solutions can be programmed to monitor (or not) any relevant variable, at the cost of possible inaccuracy or even miscalculation. One further aspect worth stressing is that RX resources can share the same hardware of the TX (or DUT in some cases), for which *lightweight* operation is a desirable property, especially in the case of scarce resources.

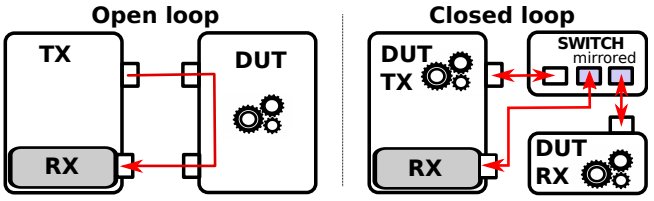

Fig. 1: Performance evaluation of a Device Under Test (DUT): open loop and closed loop

Last decade has witnessed the flourish of high speed I/O software frameworks such as Intel DPDK [1], PFQ [2], PF_Ring ZC [3], netmap [4] and PacketShader [5]. By means of (i) kernel bypassing (which circumvents the general-purpose network stack and in the meanwhile achieves zero copy), (ii) batch-processing (which amortizes the overhead of interacting with network cards and PCI buses), (iii) poll-mode packet fetching (which eliminates the overhead of interrupts), (iv) multi-queue Network Interface Controllers (NICs) combined with Receive Side Scaling (RSS) to distribute loads into multiple cores to fully exploit parallelism, these frameworks bestow general purpose hardware the ability to generate, capture and process packets in excess of 10 Gbps rates per core. This trend has resulted in the rise of software traffic generators (TX) [6]–[8], capable of saturating 10 Gbps links with minimum-size (64B) packets by occupying few cores on commodity hardware. Despite a variety of choices on TX side, this is not yet the case for software traffic monitoring on the RX side, since existing tools either offer simplistic capabilities (e.g., per-packet operations only), or require a significant amount of resources to sustain the processing of more advanced monitoring tasks, hence can hardly be co-located with TX or DUT (as detailed in Sec. II).

In this paper, we present FloWatcher-DPDK, a light software traffic monitor based on DPDK that, beyond the typical

T. Zhang, L. Linguaglossa and D. Rossi are with the Network and Computer Science Department, Telecom ParisTech, Paris, France. (e-mail: {tianzhu.zhang, linguaglossa, dario.rossi}@telecom-paristech.fr)

M. Gallo is with the Nokia Bell Labs, Nozay, France. (e-mail: massimo.gallo@nokia-bell-labs.com)

P. Giaccone is with Department of Electronics and Telecommunication, Politecnico di Torino, Turin, Italy. (e-mail: paolo.giaccone@polito.it).

per-flow statistics (e.g., throughput, flow size), is able to carry out more complex tasks, such as computing advanced statistics (e.g., percentiles) of more involved metrics (e.g., per-flow interleaving gap) while employing a minimal amount of processing resources. As an extension of our previous works [9], [10], we outline our choices in the design space (Sec. III), which we experimentally verify (Sec. IV) by systematically benchmarking FloWatcher-DPDK performance in a controlled setup (Sec. V). Our results show that by carefully exploiting the design space (e.g., reusing available flow identifiers, choosing appropriate data structures), depending on the required accuracy, packet-level and flow-level statistics incur negligible overhead (orders of magnitude smaller than the existing tools [6], [7]). Furthermore, we also perform similar evaluations with synthetic traffic and showcase FloWatcher-DPDK capabilities to monitor open- and closed-loop traffic of two open-source prototypes (Sec. VI): ClickNF [11], [12], a modular stack for the composition of L2-L7 network functions, and FD.io Vector Packet Processor (VPP) [13], a packet processing framework for software routers. We release FloWatcher-DPDK as an open source project on GitHub [14], which advances the state-of-the-art by providing fine-grained flow-level statistics at high-speed using a small amount of computation resources.

II. Related work

Relevant previous works can be categorized into high-speed traffic generation (TX), processing (DUT) and monitoring (RX). Given our monitoring focus, we refer the readers to [15], [16] for state-of-the-art on TX and DUT, respectively. In Tab. I, we show our taxonomy for existing RX frameworks in terms of achievable performance (e.g., packet and data rates), the amount of resources needed (e.g., the number of CPU cores utilized), the design choices (e.g., reusing RSS hash and the adoption of a pipeline workflow) as well as the main purpose for which the tools are designed. The literature about Speedometer, MoonGen and pktgen-DPDK does not assess the minimal number of cores required (hence we report Not Available "N.A." in Tab. I): we found it to be 2, and we will therefore use two cores for our evaluation. In order to further simplify the comprehension of the related work we divide the table into packet level and flow level depending on the monitoring operation. In this section we focus on the related work about traffic monitoring, and we purposely breakdown the available literature into *basic* vs. *advanced* tools. FloWatcher-DPDK sits in between of the two, as it attempts to perform operations pertaining to the latter class, while using as few resources as tools in the former. In particular, our goal is to reduce as much as possible the number of utilized *CPU cores*, while sustaining both wire-speed and accuracy in our measurements.

A. Basic traffic monitoring

Traffic monitoring is bundled with TX tools such as MoonGen [6] and pktgen-DPDK [7], which not only support minimum-size packet generation at line rate, but also offer basic packet-level measurement capabilities. In particular, MoonGen [6] is a high-speed traffic generator based on the DPDK framework. In addition to generation, MoonGen takes advantage of hardware features for rate control and latency measurement. Since MoonGen scripts are wrapped with the Lua programming language, the provided APIs are relatively easy to use for packet capture as well, which makes MoonGen a good candidate for both packet-level and flow-level monitoring operations. Yet, programmability in a high-level language allows for rapid prototyping of flow-based measurements at the cost of reduced performance: we complement the analysis of [6] by measuring the performance of MoonGen when flow-level operations are performed. Alternatively, DPDK-Speedometer [17] is a DPDK application capable of packet-level traffic monitoring, and can be used to measure throughput, making it a good comparison candidate for packet-level operations. These basic RX monitoring capabilities have traditionally sufficed for prototype design, especially since network functions implemented in software have long been stateless and operating on a per-packet basis. However, recent emergence of stateful and higher-level functions operating on flows (e.g., [11], [12], [23], [26], [27]) started challenging the usefulness of packet-level monitors. As such, with respect to DPDK-Speedometer or MoonGen, FloWatcher-DPDK advances the state-of-the-art by providing not only packet-level measurements, but also fine-grained per-flow statistics with negligible performance loss.

Instead of offering online statistics, a subset of the available basic traffic monitoring tools opt for storing packets for deferred processing. The work in [18] advances packet capture on commodity hardware, by complementing the well-known `pcap` library with additional features such as parallelization via RSS queues and PFQ support. Authors show that, thanks to their optimization, `pcap` is able to capture packets and perform basic counting on a 10 Gbps link with 64B packets (using at least 3 different cores). The optimized `pcap` can even be used with advanced monitoring tools such as Tstat [28], with slightly decreased performance. Similarly, the work in [20] manages to capture 300B packets at 40 Gbps line rate by exploiting a variety of techniques including Non-Volatile Memory express and Storage Performance Development Kit. FlowScope [19] is capable of continuously capturing a subset of flows that can be further dumped into disk through predefined triggers. In contrast with packet capturing tools, FloWatcher-DPDK (i) targets minimum-size packets, (ii) aims to use a smaller number of CPU cores and (iii) keeps the packet loss ratio in the order of few parts per million.

B. Advanced traffic monitoring

A set of monitoring tools with complementary capabilities have also been proposed, whose goal is to provide a more complete analysis of traffic at either local or global network scale (i.e., single or multiple points of monitoring).

Local network scale: At local level, which is the closest to this work, a number of sophisticated monitoring tools are reviewed in [21], [29]. Among them, nTop [30] and DPDKStat [21] use the least amount of resources in terms of CPU cores, despite being sophisticated tools designed for

TABLE I: Taxonomy of the related traffic monitoring work

	Proposal	Achieved Performance			Design Space		
	Name [Ref.]	Bit Rate [Gbps]	Throughput [Mpps]	Cores required	RSS Hash	Pipeline Workflow	Purpose
Packet level	Speedometer [17]	10.0	14.88	N.A.	√	√	traffic monitor
	MoonGen [6]	10.0	14.88	N.A.	√	√*	traffic generator & monitor
	pktgen-DPDK [7]	10.0	14.88	N.A.	√	×	traffic generator & monitor
	Bonelli et al. [18]	10.0	14.88	≥ 3	×	N.A.	traffic capture
	FlowScope [19]	120.0	101.35	6	×	√	traffic capture
	R.Leira et al. [20]	40.0	15.63	5	×	×	traffic capture
	FloWatcher-DPDK	10.0	14.88	2 (HT) 4 (No HT)	√	√	-
Flow level	DPDKStat [21]	40.0	$\simeq 1.59$	16	×	√	traffic monitor
	MoonGen [6]	N.A.	N.A.	N.A.	√	√*	traffic generator & monitor
	X. Wu et al. [22]	100.0	148.80	6	×	×	compare sketch algorithms
	mOS [23]	19.1	28.42	16	×	×	userspace TCP stack
	NetVM [24]	34.5	N.A.	12	×	√	NFV framework
	OpenNetVM [25]	68.0	N.A.	6	√	√	NFV framework
	FloWatcher-DPDK	10.0	14.88	2 (HT) 4 (No HT)	√	√	-

** While the out-of-the box MoonGen is in run-to-completion mode, we can set it up to work with the pipeline workflow.*

Internet traffic monitoring [28]. Although Deep Packet Inspection (DPI) and other advanced analytics can be deactivated, the TCP-based statistics are deeply entangled in both tools and would require complex modifications for deactivation. The advanced monitoring capabilities offered by these monitoring tools come at the cost of limited performance. For instance, according to [30] "using a dual core CPU, nProbe can be used for capturing packets at 1 Gbps with very little loss ($<1\%$)", whereas on similar hardware the simpler flow-level statistics tracked by our proposed FloWatcher-DPDK allow to achieve 10 Gbps with *loss rate in the order of few parts-per-million*. Similarly, DPDKStat achieves 40 Gbps processing on a NUMA system with 16 physical cores, on real traffic workload with average packet size in $[716, 811]$ bytes range corresponding to $[385, 435]$ kpps. In contrast, FloWatcher-DPDK achieves operation rates of about 7.4 Mpps per-core, *an order of magnitude* more than DPDKStat and *several orders of magnitude* with respect to Bro, Snort and Suricata [21].

To overcome the limited performance, alternative traffic monitors propose to use sketches or probabilistic summaries to analyze large datasets. The work in [22] benchmarks 3 sketch-based heavy-hitter detection methods on multi-core commodity servers and evaluates them at 100 Gbps speed. The authors consider different design choices (e.g., DPDK RX modes, data sharing scheme), and analyze them in terms of packet drop ratio, processing time and delay. Compared with FloWatcher-DPDK, this work is specifically designed for sketch-based methods which sit between packet/flow-level monitoring, thus it is not suitable for direct quantitative comparison. Integrating sketches into FloWatcher-DPDK is part of our future work.

Another class of works worth mentioning is high-speed NFV frameworks capable of composing local network monitoring functions. In particular, mOS [23] is a modular networking stack for stateful middleboxes. It provides high-level APIs with the objective of concealing the details of flow management and exposes only packet-/flow-level abstractions to make developers focus on the application logic. NetVM [24] is a high-speed NFV framework built on top of commodity hardware. It allows customizable network functions to be deployed in virtual machines. Such network functions can be chained or multiplexed to compose network elements with high flexibility. An extension of NetVM, namely OpenNetVM [25], runs network functions in Docker containers instead of VMs. It uses a manager to route packets among multiple NF chains to achieve load-balancing and high performance. All the aforementioned frameworks adopt DPDK to sustain line rate packet I/O. We argue that our work focuses on high-speed flow monitoring with low resource usage, and is orthogonal to them. FloWatcher-DPDK can either be exploited to test the performance of deployed NFs, or be integrated within an NFV network to realize per-flow monitoring functions. As such, advanced monitoring tools cannot be used for a direct quantitative comparison. Conversely, it is useful to make extensive qualitative comparison of the findings, as the operational point of simple vs. advanced traffic monitoring tools are significantly different.

Global network scale: At network level, NetFlow [31] is an example of network-level monitor with implementations in custom ASICs and pure software. A NetFlow ASIC is available only for costly high-end routers capable of dealing with up to 65k concurrent flows, whereas software solutions instead heavily rely on sampling, typically less than 1/1000 packets, which we want to avoid. Given its network-wide nature, data collection is crucial in NetFlow. As such, to reduce the size of the data to be sent to the collector, FlowRadar [32] proposes to store each flow counter using a compact data structure based on counting bloom filters. To further reduce data size, SketchVisor [33] proposes to split the data collection into regular and fast paths, which is used on load surges and only performs updates locally at the switch for a small portion of the heavy-hitter flows. sFlow [34] is another standard industry technique for network monitoring. It performs real-time packet sampling throughout L2-L7. While sFlow presents better protocol coverage and scalability, it suffers from the downside of inaccuracy due to sampling. Although NetFlow and its variants perform operations similar to ours, they differ from FloWatcher-DPDK in that the collected per-flow statistics are simpler (e.g., no per-flow percentiles) and

the data-collection stage has a prominent impact (unlike in our local case), and as such are not worth comparing directly.

III. System design

Flow monitoring can be decomposed into four stages [35], namely *packet capture* to retrieve and pre-process packets (if necessary), *flow aggregation* to aggregate packets belonging to the same flow, *data collection* to store collected statistics, and *data analysis* to provide the desired metrics characterizing different flows. In this section, we briefly review each of them and describe FloWatcher-DPDK's design space that we explore in Sec. V with the objective of providing lightweight flow-level statistics at high-speed.

A. Packet capture

The first stage consists of retrieving raw packets from the NICs, time-stamping and forwarding them to the flow monitoring engine. To capture and process packets at high speed, FloWatcher-DPDK builds on top of DPDK, a state-of-the-art packet I/O framework. We choose DPDK over alternative frameworks such as [3], [4], etc. because of its good performance, rich features, and software support, as detailed in [16]. To effectively take advantage of parallelization, we enable Receive Side Scaling (RSS) to distribute incoming traffic into multiple hardware queues, from which packets are copied into main memory without involving the CPU (Direct Memory Access) to be later processed in user-space. Besides fetching and delivering raw packets, DPDK also provides a wide range of design choices to indulge different use cases. In this paper, we only consider the most relevant ones, including programming models, multi-threading libraries as well as CPU scheduling policies, whose details are elaborated as follows:

Programming models: DPDK provides two alternative programming models, namely *run-to-completion* and *pipeline*. In the *run-to-completion* one, each packet is retrieved and processed by the same thread, typically pinned to a single core. As shown in Fig. 2-(a), each thread keeps polling its associated receive descriptor ring, fetching batches of packets into user-space for further processing. On the contrary, in the *pipeline* model, packet reception and processing logics are decoupled and distributed to separate threads. As illustrated in Fig. 2-(b), FloWatcher-DPDK pipeline model consists of two different types of threads: RX- and monitor-thread. The former keeps polling descriptor rings to deliver packets to a shared software ring. The latter fetches packets from the shared software ring for further processing i.e., collecting packet- and flow-level statistics. Since the advantages of one approach over the other are not evident, we benchmark FloWatcher-DPDK by integrating both models and select the one providing better performance while maintaining the number of used resources constant.

Multi-threading libraries: DPDK provides a novel multi-threading library, namely *lthread* [36], as an alternative to the standard POSIX one (*pthread*). Unlike pthread, lthread embraces *Cooperative Scheduling* in which a thread, instead of being preempted, periodically yields its execution to leave the chance of using CPU cycles to other threads. According

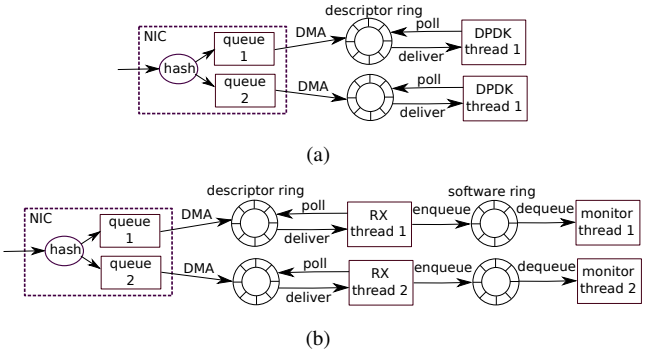

Fig. 2: Packet capture under (a) run-to-completion and (b) pipeline programming models

to [36], lthread guarantees lower scheduling overhead and provides contention avoidance. In our context where threads perform relatively simple operations (e.g., few memory accesses and instructions when hash computation is performed in software), lthread seems an appealing light-weighted solution especially if associated with pipeline programming model. Furthermore, lthread features efficient coexistence of multiple threads in a single core, which decreases the number of occupied working cores and is coherent with our goal of using exiguous processing resources. We thus implement FloWatcher-DPDK with both models and perform a selection by benchmarking their performance.

CPU schedulers: In the Linux operating system, the default CPU scheduling policy is `SCHED_OTHER`, which is based on fair time sharing and adopted by most applications. However, DPDKStat [21] uses `SCHED_DEADLINE` [37], an alternative real-time scheduling policy, to share the same cores between two threads and increase the throughput by 100%. As (i) both FloWatcher-DPDK and DPDKStat are DPDK programs able to operate in pipeline model, (ii) `SCHED_DEADLINE` enables two (or more) threads to share the same core, making our monitor occupy less resources, we therefore opt to benchmark FloWatcher-DPDK by integrating both `SCHED_OTHER` and `SCHED_DEADLINE`, and compare the performance.

Results of the considered design choices for the packet capture stage are reported and discussed in Sec. V. Although specific to FloWatcher-DPDK, we believe these benchmarking results can provide guidelines for prototype designs beyond the scope of our work.

B. Flow aggregation

After being captured, packets are aggregated into flows by means of flow identifiers (typically based on the hash of TCP-UDP/IP 5-tuple) and stored in a hash table.

Flow identification: Although it is desirable for hash functions to have good entropy properties [38], hash computation might have a non-marginal penalty in performance [39], therefore simple hash functions (sum or XOR) are generally used to compute the flow identifier [28]. We argue that extracting the 5-tuple elements from the packets incurs a non-negligible overhead (e.g., memory access, data conversion and hash computation). As the NIC already computes some

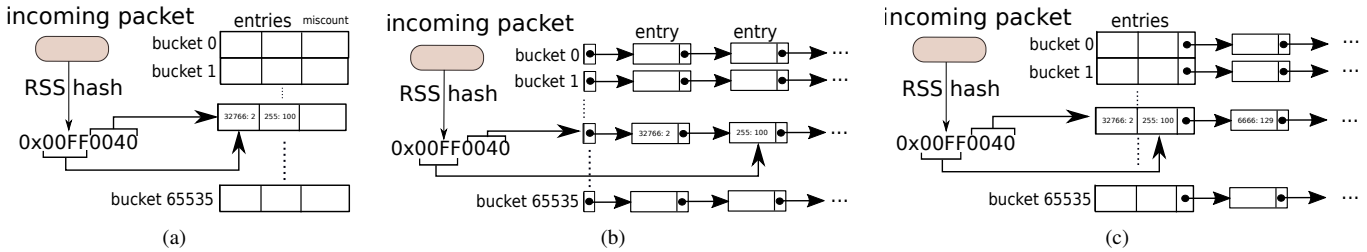

Fig. 3: Flow table implemented with (a) double open-address hash, (b) linked-list hash, (c) combined hash.

hash (i.e., Toeplitz) over the 5-tuple if Receive Side Scaling (RSS) is enabled[1], we argue that it is preferable to reuse it directly, instead of accessing & converting multiple (albeit contiguous) packet header fields over which to recompute a hash value. To validate our claim, we evaluate the overhead of data acquisition and hash computation, which are the main operations to obtain a hash, in Sec. V.

Per-flow statistics table: The value of the flow identifier is used to index the hash table storing per-flow statistics. The most naïve solution, which directly accesses an array of 2^{32} entries, requires significant amount of memory, especially given that the structure should be allocated proactively with several counters for the tasks at hand (e.g., flow size counting, flow interleaving gap, etc.). We thus consider three alternative data structures that we use to reduce FloWatcher-DPDK memory footprint, and that we (pedagogically) reported in Fig. 3.

I) *Double, open-address hash:* This data structure is shown in Fig. 3-(a). Instead of 2^{32} entries, this table contains only 2^{16} buckets, each of which consists of two static entries (or more, depending on the size of the per-flow statistics). Each entry in the table maintains a set of variables for a specific flow that are constantly updated upon packet arrivals. To reduce the number of operations, instead of using two separate hash functions as it is typically done with a double hash, we use a portion of the RSS hash (e.g., the lowest 16 bits) to address different buckets while the remaining portion (e.g., the 16 highest bits) is used to distinguish flow entries within the same bucket. The structure is efficient, as typically each bucket is cache aligned, but not accurate in the presence of several concurrent flows as only two collisions can be properly handled. Indeed, according to the birthday paradox, on average, $1.25\sqrt{2 \cdot 2^{16}} \approx 450$ concurrent flows can be tracked without collisions. To alleviate the possibility of miscounting packets, we make each bucket to contain 2 entries, each of which counts packets for a specific flow. Thus, packets indexed to the same bucket can be further distinguished according to their higher portion of hash values (e.g., the 16 highest bits).

II) *Linked-list hash:* To resolve hash collisions, one option is to enable the dynamic insertion of new entries through a linked list, as in Fig. 3-(b). This approach avoids the saturation of entries in a bucket and guarantees correct counting for all the flows. However, it suffers from the overhead of dynamic memory allocation and non-contiguous memory accesses (increased cache misses), as well as the time for the linear probing to find the appropriate element in the list. We point out that optimized data structures (e.g., red-black trees) could be used to handle chaining, making the number of memory accesses logarithmic in the worst case. Nonetheless, the payoff of these advanced data structures is low in practice since the overhead of tree management compensates most of the gain [39].

III) *Combined hash:* To simultaneously retain performance and correctness, we combine the previous two data structures, by appending the double open-address hash with a linked list in each bucket, so as to resolve *all* hash collisions, as illustrated in Fig. 3-(c). In this way, packet miscounting incurred by double open-address hash is avoided. In the meanwhile, the cache miss rate of linked-list hash is effectively reduced.

The performance of these three data structures is evaluated in details in Sec. V.

[1]In DPDK, we use TCP/IP header fields for NIC's hash computation. The 32-bit flow identifier is originally used to distribute packets among hardware queues.

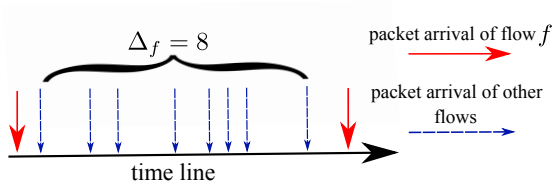

Fig. 4: Example of per-flow interleaving gap.

C. Data collection

After flow aggregation, the corresponding statistics are collected for deeper analysis and scrutiny. Data related to different flows are collected either in a volatile memory (e.g., the RAM) or in a persistent device (e.g., a SSD drive). In both cases, queries about the collected data can be performed online or offline. Unlike existing works such as nTop [30], DPDKStat [21], Netflow [31] or FlowScope [19], FloWatcher-DPDK data collection is very simple (i.e., post-mortem reports of per-flow statistics or at periodic configurable intervals). Indeed, FloWatcher-DPDK stores per-flow statistics in the volatile memory and does not export advanced features. As such, a thorough accounting of data collection in FloWatcher-DPDK is outside the scope of this paper.

D. Data analysis

FloWatcher-DPDK supports both *per-packet* and *per-flow* data analysis. At packet-level, it is capable of hardware and software packet counting. Per-packet hardware counting simply queries the NIC's registers and gets a set of basic statistics, including transmitted/received packets/bytes. Since

Algorithm 1 Update average interleaving gap $\overline{\Delta}_f$ for flow f

Require: c_f, o_f, g
1: **For each incoming packet** p:
2: f=RSS-hash(p) ▷ compute the flow identifier
3: $\Delta_f \leftarrow g - o_f - 1$ ▷ calculate current Δ_f
4: $c_f \leftarrow c_f + 1$ ▷ update number of packets for f
5: $\overline{\Delta}_f \leftarrow \frac{\overline{\Delta}_f \times (c_f - 1) + \Delta_f}{c_f}$ ▷ update average $\overline{\Delta}_f$
6: $o_f \leftarrow g$ ▷ store the old global counter
7: **end**

it is very simple and imposes nearly no processing overhead, we use it as the performance baseline that provides reference for our analysis on FloWatcher-DPDK. Per-packet software counting, on the other hand, consists of solely maintaining per-queue packet counters that are updated upon arrivals of traffic batches.

Per-flow data analysis is only available in software due to hardware limitations, and makes use of the previously outlined flow aggregation structures. Per-flow statistics can range from simple per-flow packet counting, to more complex metrics (first and second moment, percentiles of a per-flow distribution). To make an example of a fairly complex per-flow estimator, throughout this paper we consider the per-flow interleaving gap metric. In particular, for a given flow f, we define its interleaving gap Δ_f as the total number of packets received by other flows between two consecutive packet arrivals belonging to flow f. An intuitive example is illustrated in Fig. 4. Compared with per-packet time-stamping that would incur a lot of overhead, this is deemed as a cheaper solution to reflect the flows' burstiness.

The computation of the interleaving gap Δ_f is performed as follows and reported in Alg. 1. We denote g as the global packet counter, and o_f as the old global counter upon the last packet arrival of flow f. For each incoming packet p (starting from the second packet of each flow) belonging to flow f (line $1, 2$), FloWatcher-DPDK computes Δ_f (line 3), updates the packet counter of flow f (line 4), calculates $\overline{\Delta}_f$ according to Eq. (1) (line 5) and assigns g to o_f (line 6).

$$\overline{\Delta}_f = \frac{\overline{\Delta}_f \times (c_f - 1) + \Delta_f}{c_f} \tag{1}$$

To stress test FloWatcher-DPDK, instead of average or standard deviation, we monitor the *per-flow* 99*th percentile* of the interleaving gap. In our scenario, the percentile for each flow keeps fluctuating with arrivals of new packets. To appraise the percentile of many observations, a naïve method is to record all the samples and construct the cumulative distribution function (CDF). However, this method is impractical at 14.88 Mpps as it would require a significant amount of memory. Instead, we quantify the 99th percentiles of the average per-flow interleaving gaps using the well-known PSquare algorithm [40], which keeps track of a constant number of variables and requires only tens of memory accesses and few floating point operations. Our implementation of PSquare algorithm is reported in Alg. 2. Note that we omit the initial sorting operations for the first six packet arrivals

Algorithm 2 Estimate the percentile of interleaving gaps for flow f

Require: $\{q_j, n_j, n'_j, D_j\}_{j=1}^{5}, P, g, o_f$
1: $n \leftarrow \{1, 2, 3, 4, 5\}$
2: $n' \leftarrow \{1, 1 + 2 \times P, 1 + 4 \times P, 3 + 2 \times P, 5\}$
3: $D \leftarrow \{0, \frac{P}{2}, P, \frac{1+P}{2}, 1\}$
4: **For each incoming packet** p:
5: f=RSS-hash(p) ▷ extract the flow identifier
6: $\Delta_f \leftarrow g - o_f - 1$ ▷ compute the current Δ_f
7: $k \leftarrow F_p(\Delta_f)$ ▷ maximal k where $q[k] \leq \Delta_f$
8: **while** $k < 5$ **do** ▷ adjust positions
9: $n_{k+1} \leftarrow n_{k+1} + 1$
10: $k \leftarrow k + 1$
11: $i \leftarrow 0$
12: **for** $i < 5$ **do** ▷ adjust desired positions
13: $n'_i \leftarrow n'_i + D_i$
14: $i \leftarrow i + 1$
15: $q \leftarrow F_h(q, n, n')$ ▷ refer to Alg. 3
16: **end**

of each flow, for the sake of simplicity. Besides the global counters g and o_f already defined in Alg. 1, PSquare algorithm requires 15 variables, namely $\{n_j, q_j, n'_j\}$ for each of the 5 per-flow markers $j = 1..5$. We define n_j as the real positions of the 5 markers, q their heights, n' their desired positions and P the target percentile. $\{q_j\}_{j=1}^{5}$ correspond to the minimum observed interleaving gap ($q_j = 1$), current estimation of $\frac{P}{2}$-($q_j = 2$), P- ($q_j = 3$), $\frac{P+1}{2}$-percentiles ($q_j = 4$), and the maximum observed interleaving gap ($q_j = 5$). According to the description in [40], we initialize n and n' as the values shown in the initialization part of Alg. 2 (line 1-2). D denotes an array of constant values used to adjust the desired markers' positions (initialized at line 3). For each incoming packet p belonging to flow f, we compute the current interleaving gap Δ_f (line $4 - 6$). F_p is a simple function to return the highest index k such that $q[k] \leq \Delta_f$ (line 7). Based on the output of F_p, we update the positions of the 5 markers (line $8 - 10$) and adjust positions of n' accordingly (line $11 - 14$). The last step is to update the heights of the 5 markers q using the F_h function (line 15), which is elaborated in Alg. 3. According to [40], the $2 - 4$ markers need to be adjusted if their positions are off to the desired ones by more than 1 unit. Therefore, for each of the $2 - 4$ markers, we compute the distance from its desired position (line $1 - 3$). If the absolute distance is more than 1 unit (line $4 - 5$), the height of this marker is adjusted in accordance with a piecewise parabolic prediction formula (line 6). If this height is coherent in ascending order with other markers, it is saved as the new height (line $7 - 8$); otherwise, the height is updated using linear formula instead (line $9 - 10$). Then we adjust once again the marker's position (line 11). After inspecting the heights of all the $2 - 4$ markers, F_h returns the adapted heights of the markers (line 13).

After executing the described algorithms, the value of marker q_3 is the estimated Pth percentile of Δ_f. The source code of the PSquare algorithm used in FloWatcher-DPDK is

Algorithm 3 F_h: Update the heights of the 5 markers for the PSquare algorithm

Require: $\{q_j, n_j, n'_j\}_{j=1}^{5}$

$i \leftarrow 1$
for $i \leq 3$ **do** ▷ for $2-4$ markers
  $d \leftarrow n'_i - n_i$ ▷ Compute distance
  if $(d > 1$ and $n_{i+1} - n_i > 1)$ or $(d \leq -1$ and $n_{i-1} - n_i < -1)$ **then**
    $d \leftarrow (d > 0)\ ?\ 1 : -1$
    $q' \leftarrow q_i + \frac{(n_{i+1}-n_i-d)\times(q_i-q_{i-1})}{n_i-n_{i-1}} + \frac{d}{n_{i+1}-n_{i-1}} \times \frac{(n_i-n_{i-1}+d)\times(q_{i+1}-q_i)}{n_{i+1}-n_i}$
    if $q' > q_{i-1}$ and $q' < q_{i+1}$ **then**
      $q_i \leftarrow q'$ ▷ parabolic formula
    else
      $q_i \leftarrow q_i + \frac{d\times(q_{i+d}-q_i)}{n_{i+d}-n_i}$ ▷ linear formula
    $n_i \leftarrow n_i + d$ ▷ adjust position
  $i \leftarrow i + 1$
return q

available in [41], whose correctness has been verified with data provided in the original paper [40]. The performance of FloWatcher-DPDK with the integration of PSquare algorithm is reported in Sec V-C.

By integrating the PSquare algorithm, our software traffic monitor is able to estimate the percentile of the distribution of the interleaving gaps for each monitored flow without storing all the observations. PSquare algorithm requires significant computation (calculating Δ_f, n, n' and q for each received packet) and memory accesses (fetch the value of n, n', q) to estimate the percentiles. Therefore, while significantly simpler than the average tasks performed by advanced monitoring tools such as DPDKStat [21], the estimation of the 99-th percentile through the PSquare algorithm represents a *stressful* scenario for RX tools that aim at using a limited amount of computation resources.

IV. Testbed environment

In this section, we provide hardware and software configurations details used in our experiments. We also introduce TX side settings, including traffic workload and patterns.

A. Hardware

To better validate our results, we use three testbeds with different hardware setups:

Testbed 1: Two commodity servers, each of which is equipped with 2 Intel Xeon 2.60GHz CPUs (each with 20 physical cores), using 32k/256k/25600k L1-3 caches, and 1 Intel 82599ES dual-port 10-Gbps NICs (subsystem: Hewlett-Packard Company Ethernet 10Gb dual-port 560FLR-SFP+ Adapter). Both servers run a Linux 4.4.0-based distribution.

Testbed 2: A commodity server running a Linux 4.8.0-based distribution, equipped with 2 Intel Xeon E5-2690 v3 @ 2.60GHz CPUs (each with 24 physical cores), using 32k/256k/30720K L1-3 caches, and 2 Intel 82599ES dual-port 10-Gbps NICs (subsystem: Intel Corporation Ethernet Server Adapter X520-2).

Testbed 3: This commodity server runs a Linux 4.15.0 kernel and is equipped with 2 Intel Xeon CPU E5-2650 v3 @ 2.30GHz (each with 20 physical cores), using 32k/256k/25600k L1-3 caches, and Intel 82599ES dual-port 10-Gbps NICs (subsystem: Intel Corporation Ethernet Server Adapter X520-2).

For each testbed, the traffic generator (TX) is directly connected to FloWatcher-DPDK. Such configuration allows us to precisely measure the amount of losses due to the RX, without incurring losses caused by packet processing at intermediate DUTs. According to [13], we configure the servers to isolate cores for the exclusive usage of FloWatcher-DPDK. The CPU frequency scaling governors are set from "on demand" to "performance" for all the active cores to maximize the processing speed. We ran our tests (without integrating PSquare algorithm) on all three testbeds, and only observed packet loss on Testbed 1. This paper mainly shows *conservative* results obtained on it. A sensitivity analysis of the performance discrepancy between the three testbeds is discussed in Sec. VI-B.

B. Software

For our tests, we use MoonGen as TX, FloWatcher-DPDK as RX, and experiment with the different design choices outlined earlier. As main RX performance metric, we compute the Packet Drop Ratio (PDR), i.e., the average drop probability; we consider PDR to be the primary metric to assess RX performance, as losses at the monitor alter the accuracy of any other flow-level metrics. After having accurately estimated the PDR, in Sec. VI we additionally consider the throughput with ClickNF [12] and VPP [13] deployed as DUT, to showcase the practical usage of FloWatcher-DPDK.

C. Traffic workload

For the initial experiments to evaluate the performance of FloWatcher-DPDK, we adopt open-loop traffic. In particular, MoonGen (TX) is used to generate synthetic traffic. Under synthetic traffic (Secs. V, VI-A), all packets are 64B and injected at 10 Gbps (which corresponds to 14.88 Mpps). A flow is identified by the standard TCP/IP 5-tuple and the identifiers are generated according either to a uniform distribution or a Zipf law (with $\alpha = 1$) across a set of 2^{16} flows, corresponding to a hash load factor of 1.0. To measure the loss rate with high precision (reported in a 10^{-6} scale, i.e., part-per-million), each experiment lasts around 6 minutes (generating and processing approximately 5.2 billion packets), and all the graphs show the average and 95% confidence interval on 50 repetitions of the same test. For the final experiments with open source prototypes, in Sec. VI-C we adopt closed-loop DUT traffic, generated with a fast TCP server/client implementation. In particular we use ClickNF [11], [12], a modular network L2-L7 stack for end-host and middlebox functions. A given pool of ClickNF clients connect to the ClickNF server using random TCP source ports in the range $[1, 65535]$, i.e., 65k flows as in the previous scenarios. Once the connection is established, the

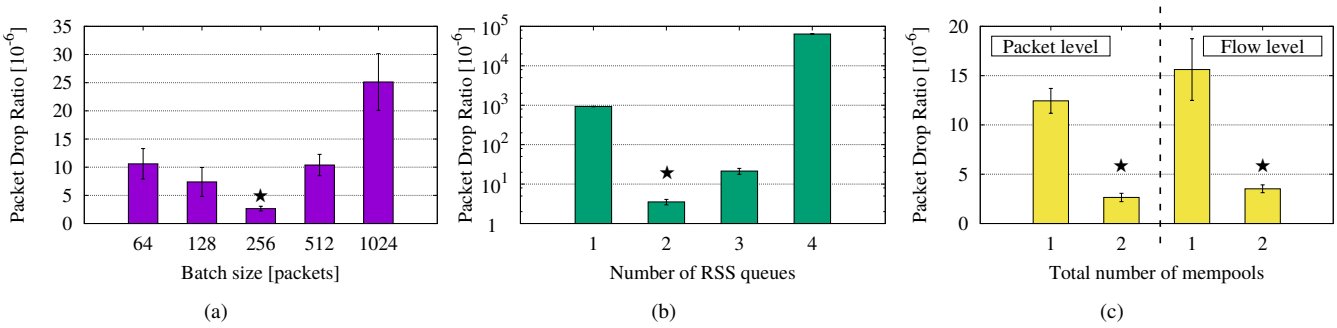

Fig. 5: Impact of (a) batch size (b) number of RSS queues, and (c) of mempools. *Optimal* parameters are denoted with $\star$.

client sends a 512B packet that is echoed back by the server. When the client receives the echo message, the connection is closed and the procedure repeated.

As for open-loop traffic, in Sec. VI-C we consider a software router using the stack provided by the FD.io Linux Foundation project [42]. In particular, we consider the case in which the VPP router employs a per-flow scheduling algorithm to enforce bandwidth fairness among flows [27]. Packets belonging to flows exceeding their fair rate are dropped while packets belonging to flows sending less than their fair rate are forwarded without any drop. The traffic is again injected by MoonGen (TX) at full rate, and the number of concurrent flows is 1000 with variable rates, where the rate of the k-th flow is k times slower than the first one (i.e., the fastest).

V. Software tuning

As FloWatcher-DPDK aims at using a small amount of CPU resources, low-level parameters play an important role in the overall performance. In this section, we carefully validate our system by contrasting the different alternatives. We begin by conducting a set of experiments to tune the *packet capture* stage, and then detail the performance of the *flow aggregation* and *data analysis* stages. Tab. II reports values yielding the best performance i.e.,"*tuned*" that are hence adopted unless otherwise specified.

A. Packet capture stage

In the *packet capture* stage, there are several relevant parameters to tune, including the batch size, the number of HW queues and the configuration of memory pool. Since this is the initial stage of packet processing, we also detail the optimal design choices such as the programming models and CPU schedulers. To tune the parameters in this stage, we configure MoonGen to generate 64B packets at 14.88 Mpps rate, and run FloWatcher-DPDK to perform packet monitoring at both packet-level (software packet counting) and flow-level (per-flow packet counting), whose PDRs are reported respectively. By varying one parameter while fixing the others, we derive the set of *optimal* ones.

TABLE II: *Optimal* FloWatcher-DPDK parameters

Parameter	Value
Flow table structure	double hash
RX queue size	4096 packets
Batch size	256 packets
Number of RSS queues	2 queues
Number of cores	4 cores (no HT)
Number of mempools	one for each RSS queue
Hyper-threading	disabled
Programming model	pipeline with pthread
Flow analytics	per-flow packet counting
CPU scheduler	SCHED_OTHER

Batch size: The batch size is the maximum number of packets fetched through one poll-mode query in DPDK. Similar to [43], we evaluate the impact of batch size ranging from 64 to 1024 packets. With small batches FloWatcher-DPDK cannot properly handle RX bursts and this leads to losses in the NIC, whereas too large batches may take longer to process thus also yield to losses in the RX. Fig. 5-(a) shows that losses are minimal for 256 packets batches – an operational point at which the Direct Memory Access (DMA) operations are likely optimized and that we adopt for the following experiments.

Number of HW queues: By enabling Receive Side Scaling (RSS), the load of incoming traffic is distributed into multiple queues, associated with different threads (or cores). While increasing the number of RSS queues is beneficial for load balancing purposes, it can lead to severe contention on the PCI bus, which in turn degrades the overall performance [44]. Fig. 5-(b) reports the PDR with increasing the number of RSS queues, from which we infer that 2 queues are *optimal* to handle worst-case scenario on our testbed (we thus use 2 RSS queues in what follows). Conversely, given the lightness of our tasks at hand, increasing the number of RSS queues further yields to performance degradation due to increased PCI bus contention. Note that this phenomenon does not appear in tools performing more complex analytics [21] on bigger packets, where CPU is a more stringent bottleneck.

Per-queue memory pool: DPDK reserves descriptor pools to associate incoming packets, avoiding the overhead of runtime memory allocation. Most of the DPDK sample applications use a *single* memory pool for multiple queues, which might increase contention, even in the presence of per-core mempool cache. An interesting point to check is whether the performance can be improved by allocating a separate mempool for each RSS queue. As shown in Fig. 5-(c), our intuition proves correct for both per-packet and per-flow operations. Per-queue mempool is heavily beneficial in both cases, reducing the loss rate by roughly one order of magnitude, and we thus only consider per-queue mempool henceforth.

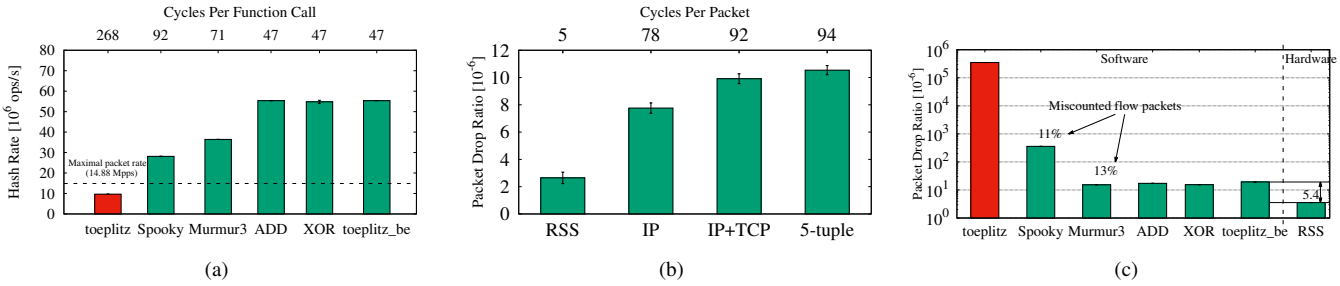

Fig. 6: Performance dissection of online hash computation: (a) absolute computation rates of different hash functions; (b) incurred PDR of memory access and data conversion when receiving packets at 14.88 Mpps, (c) Average PDR with online hashing. "RSS-HASH" represents the reference PDR by reusing the pre-computed RSS hash of the packet descriptor.

Programming models: We implement two flavors of FloWatcher-DPDK following different programming models. The first one adopts the run-to-completion model while the other is based on the pipeline model. As shown in Tab. III, the pipeline model outperforms the run-to-completion one (for the sake of space, we report only results for POSIX thread). With the pipeline model, the retrieved packets are staged in a big software ring before being processed by the monitoring threads: the software ring acts as a buffer that absorbs the processing delays of packet monitoring threads, thus achieving higher throughput. We thus adopt the pipeline model and all the results of FloWatcher-DPDK shown in the following sections are based on it.

CPU schedulers: As discussed in Sec. III-A, we implement two versions of FloWatcher-DPDK using `SCHED_OTHER` and `SCHED_DEADLINE` respectively. Both versions adopt pipeline model with 2 RSS queues, namely two pairs of (RX, monitor) threads. Note that we enable `SCHED_DEADLINE` according to the instructions in [37]. Each pair of (RX, monitor) threads is pinned to the same core, by mapping logical cores to physical ones. As shown in Tab. III, FloWatcher-DPDK using `SCHED_DEADLINE` loses approximately three orders of magnitude more packets than the case of using the default `SCHED_OTHER` policy. Hence, even if `SCHED_DEADLINE` manages to boost performance for DPDKStat [21], it fails to do so for FloWatcher-DPDK. We thus consider `SCHED_OTHER` as the *optimal* scheduling policy for FloWatcher-DPDK.

B. Flow aggregation stage

For the flow aggregation stage, we begin by quantifying the performance gain of directly utilizing RSS hash available inside packet descriptors. Then we characterize the performance of the three data structures for hash table described in Sec. III-B and discuss other implementation dtails.

TABLE III: Packet drop ratio for different programming models and CPU schedulers. A $\star$ sign denotes *optimal* parameters.

Parameter	Average PDR [$\times 10^{-6}$]	95% C.I.
Run-to-completion	4.8	0.57
Pipeline with pthread $\star$	2.6	0.56
SCHED_DEADLINE	2500	300
SCHED_OTHER $\star$	2.6	0.56

Using RSS hash: For evaluating the cost of hash computation, we choose 6 state-of-the-art hash functions and measure their absolute rates. In particular, we select DPDK's toeplitz and its enhanced toeplitz_be [45] version. Since these functions are used by the NIC to compute RSS hash, their achievable throughput can reflect the overhead directly. In addition, we consider other hash functions, including simple addition (ADD), exclusive OR (XOR), SpookyHash [46] (Bob Jenkins) and Murmur3 [47]. The absolute hash rates of the considered functions are illustrated in Fig. 6(a). We also report, on the top X-axis, the average number of CPU cycles consumed during a single function call. In particular, toeplitz fails to run beyond 10 million/s due to its complexity, making it unpractical for on-line processing (i.e., 14.88 Mpps at 10Gbps). Other hash functions, even though with different performances, can be potentially used to compute flow identifiers at line rate. Furthermore, all functions consume extra CPU cycles (around [47 - 268] cycles/call), which can be saved if the RSS hash computed by the NIC is used.

Another source of overhead while computing the hash in software is accessing memory to read packets' headers and retrieve the 5-tuple fields. To evaluate this overhead, we configure FloWatcher-DPDK to access packets' headers, convert their data formats to CPU byte order and compute the resulting rate. In particular, we consider the following cases for flow identification: (i) reuse RSS hash, (ii) IP (use IP src/dst addresses, 8 bytes), (iii) IP + TCP (in addition to the previous case, use TCP src/dst ports, 12 bytes) and (iv) 5-tuple (use all the 5-tuple fields, 13 bytes). The packet loss ratio and number of consumed cycles per packet are illustrated in Fig. 6(b). Note that there is a sharp increase of the packet drop ratio ($\times 3.98$) between RSS and 5-tuple, this is mainly due to the overhead caused by memory access. Although all the considered cases incur low Packet Drop Ratio (PDR), accessing memory and converting header fields already consume up to 100 cycles/packet, which can be avoided using RSS hash value that is already available in DPDK metadata.

Finally, we further quantify RSS efficiency in the context of online high-speed packet reception (i.e., RX) by evaluating PDRs. For online hash computation, we extract the 5-tuple from incoming packets and calculate the 32-bit hash value. The hash is then used as flow identifier to index the entries in the flow table. Similar to the previous tests, FloWatcher-

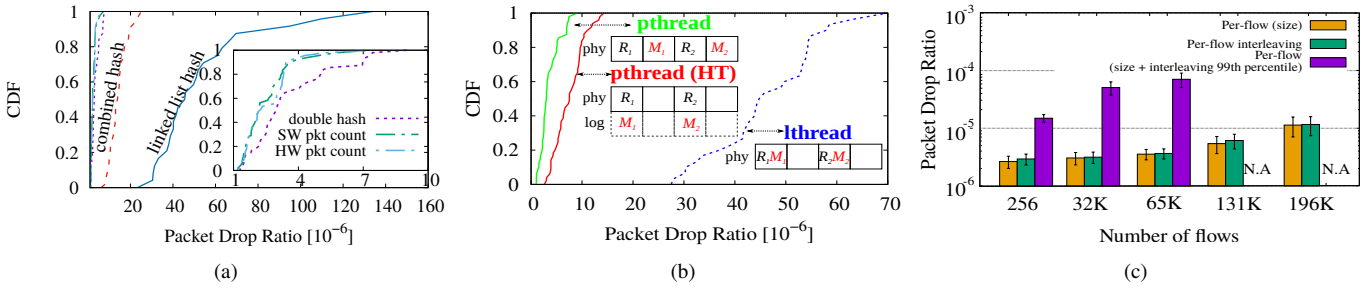

Fig. 7: Design choices of packet aggregation stage: (a) CDF of packet drop ratio for different data structures, (b) CDF of packet drop ratio for configurations for the combined structure. Data analysis stage: (c) CDF of packet drop ratio for performance comparison for simple (per-flow packet counting/interleaving gap) and complex (per-flow 99th percentile of the interleaving gap) statistics with increasing number of flows

DPDK is configured to receive 64B packets at 10 Gbps line rate for 6 minutes. Experiment for each function is repeated 10 times. The results for all the functions are detailed in Fig. 6(c), with the approach of reusing the RSS hash depicted here as a reference. As we can observe, DPDK's toeplitz hash function incurs a PDR of 30%, which is coherent with our previous test of absolute hash rate (10 Mpps vs. 14.88 Mpps). Murmur3 has the minimal PDR among the hash functions we compare, yet it incurs a flow miscounting rate of 11%. Similarly, Spooky hash also incurs 13% of flow miscalculation. We believe such miscounting can be avoided by carefully choosing the seed for each hash function, but this is outside the scope of our work. Add, XOR and toeplitz_be all incur PDRs in $\approx 10^{-5}$, which is small but still around 5 times worse than the case of RSS. Note that simple functions such as ADD and XOR are not suitable for flow aggregation under more complicated traffic patterns due to potential hash collisions. Toeplitz_be, as illustrated in Fig. 6(c), loses more than 5.4 times the packets compared with RSS. Based on the observations above, we argue that FloWatcher-DPDK presents the best performance (in terms of PDR) by exploiting the available RSS hash already computed by the NIC.

Data structure: Fig. 7(a) compares the performance of FloWatcher-DPDK with different data structures and shows HW/SW packet counting as baseline references. Per-flow SW counting performance is significantly affected by different data structures. In particular, losses of double hash are comparable with those of SW packet counting, although double hash can incur miscounting and should be avoided in case of huge number of flows on the hash table. Conversely, single hash with linked lists offers precise counting but incurs a non-negligible overhead due to non-contiguous memory allocation. This translates into a precision decreased by orders of magnitude for the same amount of CPU resources, and should thus be avoided in case of high input rates.

Finally, combined hash sits at the intermediate point in the performance tradeoff, achieving precise counting for a limited overhead. Per-packet SW and HW losses are very close, and provide a lower bound for the PDR of flow-level measurement, settling around a measurable $2{\cdot}10^{-6}$ on average. It should be noted that packet losses with HW, SW packet counting are negligible[2] and mainly due to software generation inaccuracy [15], [48].

Multithreading implementation: The performance of flow aggregation stage also depends on the selection of libraries for thread implementation. Considering the double-hash data structure (for the sake of simplicity) and limitedly focusing on the pipeline model (similar results holds for run-to-completion model), we compare the *lthread* vs. *pthread* implementations. As explained in Sec. III-A, the pipeline model consists of two different types of threads, an RX-thread (R) and a monitor-thread (M), for each pipeline, i.e., for 2 RSS queues, 4 distinct threads (R_1, R_2, M_1, M_2) in total. The binding between threads and cores i.e., pinning, is illustrated in Fig. 7(b). Note that each cells in the first row represents a physical core, while each cell in the second row represents a logical/virtual core, usually enabled by hyper-threading. In case of lthread, cooperative multi-threading is implemented within the application itself: R and M threads of each processing pipeline coexist on the same core and the application manages their execution. In case of pthread, threads can be pinned to different physical cores, or to two logical cores of the same physical one.

We observe that, with respect to lthread, pthread significantly decreases the packet losses; at the same time, there seems to be a further advantage into separating the receiver and the monitoring threads across multiple cores, which is in stark contrast with more complex monitoring software such as DPDKStat [21]. A plausible explanation is that complex software requires to perform more memory accesses: in this case, hyper-threading is beneficial to keep the pipeline full when the execution of another pipeline is stalled. Conversely, in our case the lower memory usage, coupled to a cache-friendly memory structure, may diminish the usefulness of hyper-threading (similar phenomena are also observed in optimized VPP software stack [13] with high instruction-per-clock efficiency).

To conclude, we note that even though lthread configuration loses nearly one order of magnitude more packets than its pthread counterpart, it can still achieve 14.88 Mpps throughput with only 2 cores. Therefore, we believe lthread implementa-

[2]A loss probability of $2 \cdot 10^{-6}$ corresponds to about 30pkts of size 64B per second, i.e., a throughput distortion <16 kbps for a full 10 Gbps TX-rate.

tions can still serve equal purpose with tolerable packet loss (e.g., 40 losses per million).

C. Data analysis stage

Finally, we take into account the impact of increasing the number of flows and two types of analytics: simple per-flow packet counting (only per-flow packets and byte counters are maintained) and the more complex per-flow 99th percentile of the interleaving gap which requires several memory accesses and state manipulations, as we extensively explained in Sec. III.

Results are shown in Fig. 7(c), where FloWatcher-DPDK is used to evaluate simple and complex analytics over an increasing number of flows. As the number of flows increases, so does the hash load, and thus the penalties due to the linked list. It can be observed that the cost of per-flow percentile estimation increases the losses by an order of magnitude: at the same time, the distortion remains below 10^{-5} for load up to 1 (i.e., $65k$ flows for a double hash of $65k$ entries), which may yield to tolerable distortion in many use cases. Conversely, in case of simple analytics, PDR remains on average below 10^{-5} for hash loads up to 1, achieved with only 2 cores. Note that for higher hash load ($> 65k$), then the losses are too high for the targeted scenarios, and hence omitted.

VI. Experimental results

In this section, we test FloWatcher-DPDK with different scenarios. Specifically, Sec. VI-A contrasts the performance of FloWatcher-DPDK to that of other tools (using their optimal configuration) under synthetic traffic. Next, Sec. VI-B investigates the results obtained over different testbeds highlighting the reason for packet losses in Testbed 1. Finally, Sec. VI-C considers two operational scenarios in which we employ our tool to assist the evaluation of closed-loop and open-loop DUT (respectively ClickNF and VPP).

A. Traffic monitor precision

Synthetic traffic: Fig.8 shows the performance for hardware packet counting (left, orange) vs. software packet counting (center, yellow) vs. software per-flow monitoring (right, violet). For all the chosen tools, we additionally report "default" and "tuned" settings. In the "default" ones, each tool is used unmodified, while in the "tuned" ones, we employ the same *optimal* parameters chosen for FloWatcher-DPDK as described in Sec. V, when applicable (i.e., we used the same batch size, number RSS queues, Hyperthreading, etc. but did not change the DPDK programming model as it frequently requires a complete redesign of the software). It is important to notice that MoonGen, Speedometer and pktgen-DPDK are all used with two physical cores: in fact, increasing the number of cores does not bring any advantage to the packet drop ratio. On the contrary, FloWatcher-DPDK can be used with either two physical cores (four threads mapped to two physical cores in hyperthreading) or four physical cores (four threads mapped to different cores). In this case, in order to obtain the minimal packet drop ratio we need to assign four physical cores. However, even the configuration with two physical cores in hyperthreading is sufficient to provide a packet drop ratio which is worse than the previous scenario, but better than the state-of-the-art. At the same time, the two-core configuration reduces even more the amount of resources needed. Experiments under uniform and Zipf distribution of the flows are reported respectively. Hardware packet counting is reported as a reference. In particular, packet-level counting performance are reported for Hardware, MoonGen, Speedometer, pktgen-DPDK and FloWatcher-DPDK. As illustrated in Fig. 8, FloWatcher-DPDK exhibits at least two orders of magnitude lower PDR with respect to MoonGen and pktgen-DPDK, which approximately reflects the overhead of wrapping C code with Lua language. This is mainly due to the careful design and parameter tuning of the packet capture stage described in Secs. III and V respectively. Intuitively, FloWatcher-DPDK performs similarly to Speedometer because of their similarities when consider packet-level counting statistics.

At last, we consider per-flow counting, contrasting FloWatcher-DPDK with a custom Lua implementation of the double hash (without linked list) strategy in MoonGen, taking advantage of a recent MoonGen feature that allows to directly access the RSS hash computed by the NIC directly in Lua (thus also without hashing and memory access overhead). Two main takeaways can be derived from the picture with this respect. First, activating per-flow counting in MoonGen increase losses by one or two orders of magnitude depending on the scenario. Second, activating per-flow counting in FloWatcher-DPDK has only a negligible effect as far as packet loss is concerned. We believe such performance gap is mainly due to the overhead of the Lua scripting language and to the careful design and parameter tuning of the four flow monitor stages described in Secs. III and V.

B. Sensitivity analysis

We now investigate the packet loss observed in Testbed 1. We specifically discuss the difference with the other two testbeds, which do not suffer from packet losses when running the same set of tests (if we exclude the case of PSquare algorithm, which is more computationally expensive, and leads to some losses in Testbeds 2 and 3 as well). The main discrepancies in terms of hardware configuration lie in distinct CPU models, L3 cache sizes, and network cards (all servers employ Intel 82599 controller but the cards come from with different manufacturers).

We use Perf [49] (version 4.8.17), a state-of-the-art performance analysis tool in Linux providing statistics on CPU cache misses, CPU cycles, etc. to profile FloWatcher-DPDK on the different testbeds. To characterize the impact of cache size and CPU model, we choose CPU cache miss rate and

TABLE IV: FloWatcher-DPDK profiling for different testbeds

	Testbed 1	**Testbed 2**	**Testbed 3**
Cache miss rate [%]	0.008	0.011	0.014
Instructions per cycle	2.030	1.810	2.080
Packet loss ratio (PDR)	3.5×10^{-6}	0	0

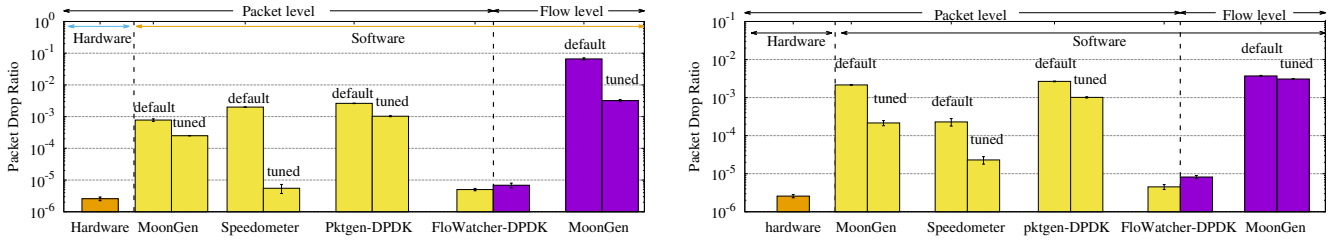

Fig. 8: Traffic monitoring: performance under Synthetic traffic with (a) Uniform and (b) Zipf patterns

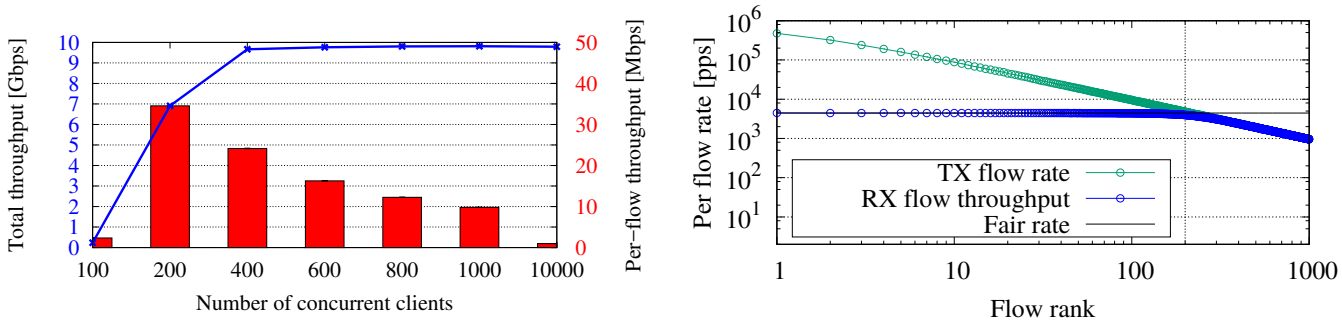

Fig. 9: DUT testing: (a) closed-loop transport function in ClickNF (with the line points representing the total throughput in Gbps and the histogram representing the per-flow throughput in Mbps) and (b) open-loop scheduling function in VPP.

instruction per cycle as main metrics. The profiling results of Perf when FloWatcher-DPDK is used for packet-level counting (i.e., same test executed in Sec.VI) are listed in Tab. IV.

As we can observe, on Testbed 1, FloWatcher-DPDK incurs lower cache miss rate and comparable instruction rate, with respect to the other two. So neither CPU model nor cache size is the main contributor to packet losses. Thus, we derive that the packet losses are due to Testbed 1's network card, produced by a different manufacturer with respect to the other testbeds.

C. Traffic monitor usage

After evaluating the PDR, we proceed to demonstrate the practical usage of FloWatcher-DPDK considering two use cases namely ClickNF and vpp closed- and open-loop.

ClickNF transport: Fig. 9-(a) shows the per-flow (bars) and total throughput (line) measured in the ClickNF client-server scenario described in Sec. IV-C in which clients connect to a ClickNF server using random TCP source ports in order to send a single 512B packet that is echoed back by the server. Surprisingly, when the number of concurrent clients is small (i.e., < 400), total and per-flow throughput are far from line rate. This can be explained by the fact that, in this scenario, ClickNF cannot completely exploit the performance bonuses given by batch processing and poll mode, due to the low amount of packets that need to be transferred. Indeed it is known [50] that, when processing small batches, DPDK exhibits poor performance mainly due to congestion on the PCIx bus (the same behavior can be observed with small batch size, as discussed in Sec. V). Conversely, when the number of concurrent clients is bigger (i.e., > 400), ClickNF throughput achieves close to line rate (i.e., 9.8 Gbps) and FloWatcher-DPDK provides a handy complement to the ClickNF log, allowing to check fairness and efficiency of the TCP server implementation.

VPP scheduling: We use FloWatcher-DPDK to monitor both the TX and the DUT traffic, achieving the performance reported in Fig. 9-(b). It can be noticed that (i) the first 200 most aggressive flows observe a decrease in their rate because they exceed their fair share of the link, while (ii) the output rates of the remaining flows match their input ones, since their rates are lower than their fair share. As such, the per-flow fair sharing mechanism implemented in [27] operates as expected. Note that, to reliably monitor low-rate flows, and thus check the correctness of the implementation, the accuracy of FloWatcher-DPDK is of uttermost importance.

VII. Conclusions

In this paper we design, tune and experiment FloWatcher-DPDK, a high-speed lightweight software traffic monitor capable of providing fine-grained per-packet and per-flow statistics at 10 Gbps line rate with 64B packets, using only 2 CPU cores. In particular, FloWatcher-DPDK manages to conduct more complex tasks (e.g., per-flow interleaving gap, percentile, etc.) with tolerable packet loss. The tool leverages the RSS hash, which is computed beforehand by the NIC and available as packet metadata, as flow identifier to avoid extra computation and memory access, and employs a careful design where flow-tables are aligned with cache line boundaries. Thanks to its careful design and to the extensive parameter tuning we performed, FloWatcher-DPDK outperforms state-of-the-art alternative solutions and provides researchers with a precise,

flexible, and open-source tool for monitoring the performance of their high-peed prototypes. Moreover we believe parameter tuning and design choices can also provide guidelines for implementing other applications in the high-speed domain. Finally, to demonstrate its usage, we further expound two use cases in which FloWatcher-DPDK is used to derive the performance of other open source prototypes. We make the tool available at [14]. Although we only consider 10 Gbps monitoring rate at this moment, we argue that it is only a matter of horizontal expansion of CPU cores to reach 40 Gbps or beyond, and in this paper we focus on traffic monitoring with low usage of resources.

ACKNOWLEDGMENTS

This work has been carried out at Laboratory of Information, Networking an Communication Sciences (LINCS) and benefited from the support of NewNet@Paris, Cisco's Chair "NETWORKS FOR THE FUTURE" at Telecom ParisTech. (http://newnet.telecom-paristech.fr).

REFERENCES

[1] http://dpdk.org/.
[2] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On multi–gigabit packet capturing with multi–core commodity hardware," in *PAM*. Springer, 2012.
[3] https://github.com/ntop/PF_RING.
[4] http://info.iet.unipi.it/~luigi/netmap/.
[5] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, 2010, pp. 195–206.
[6] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *ACM Conference on Internet Measurement Conference*, 2015, pp. 275–287.
[7] Intel Pktgen-DPDK. https://github.com/pktgen/Pktgen-DPDK.
[8] TRex: Realistic traffic generator. https://trex-tgn.cisco.com/.
[9] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, and D. Rossi, "FlowMon-DPDK: Parsimonious per-flow software monitoring at line rate," in *2018 Network Traffic Measurement and Analysis Conference (TMA)*. IEEE, 2018, pp. 1–8.
[10] ——, "High-speed per-flow software monitoring with limited resources," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. ACM, 2018, pp. 138–140.
[11] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi, "Climb: Enabling network function composition with Click middleboxes," *ACM SIGCOMM Computer Communication Review*, 2016.
[12] M. Gallo and R. Laufer, "ClickNF: a modular stack for custom network functions," in *USENIX Annual Technical Conference*, Boston, MA, 2018, pp. 745–757.
[13] https://wiki.fd.io/view/VPP.
[14] https://github.com/ztz1989/FloWatcher-DPDK.
[15] P. Emmerich, S. Gallenmüller, G. Antichi, A. W. Moore, and G. Carle, "Mind the gap: A comparison of software packet generators," in *ACM/IEEE ANCS*, 2017.
[16] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *ACM/IEEE ANCS*, 2015.
[17] https://github.com/hpcn-uam/.
[18] N. Bonelli, S. Giordano, and G. Procissi, "Enabling packet fan-out in the libpcap library for parallel traffic processing," in *IEEE TMA*, 2017.
[19] P. Emmerich, M. Pudelko, S. Gallenmüller, and G. Carle, "FlowScope: Efficient packet capture and storage in 100 Gbit/s networks," in *International IFIP TC6 Networking Conference*, 2017.
[20] G. Julián-Moreno, R. Leira, J. E. L. de Vergara, F. J. Gómez-Arribas, and I. González, "On the feasibility of 40 Gbps network data capture and retention with general purpose hardware," in *ACM Symposium on Applied Computing*, New York, NY, USA, 2018, pp. 970–978.
[21] M. Trevisan, A. Finamore, M. Mellia, M. Munafò, and D. Rossi, "Traffic analysis with off-the-shelf hardware: Challenges and lessons learned," *IEEE Communications Magazine*, vol. 55, no. 3, pp. 163–169, 2017.
[22] X. Wu, P. Li, Y. Ran, and Y. Luo, "Network measurement for 100 GbE network links using multicore processors," *Future Generation Computer Systems*, vol. 79, pp. 180–189, 2018.
[23] M. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mOS: A reusable networking stack for flow monitoring middleboxes," in *USENIX NSDI*, Berkeley, CA, USA, 2017, pp. 113–129.
[24] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.
[25] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "OpenNetVM: A platform for high performance network service chains," in *ACM Hot topics in Middleboxes and Network Function Virtualization*, 2016, pp. 26–31.
[26] G. Liu, Y. Ren, M. Yurchenko, K. Ramakrishnan, and T. Wood, "Microboxes: High performance NFV with customizable, asynchronous TCP stacks and dynamic subscriptions," in *ACM Special Interest Group on Data Communication*, 2018.
[27] V. Addanki, L. Linguaglossa, J. Roberts, and D. Rossi, "Controlling software router resource sharing by fair packet dropping," in *IFIP Networking*, 2018.
[28] A. Finamore, M. Mellia, M. Meo, M. M. Munafo, and D. Rossi, "Experiences of Internet traffic monitoring with Tstat," *IEEE Network Magazine*, vol. 25, no. 3, pp. 8–14, May 2011.
[29] Y. Zhou, O. Alipourfard, M. Yu, and T. Yang, "Accelerating network measurement in software," *ACM SIGCOMM Computer Communication Review*, vol. 48, no. 3, 2018.
[30] https://www.ntop.org/.
[31] B. Claise, G. Sadasivan, V. Valluri, and M. Djernaes. (2004) RFC3954 Cisco Systems NetFlow Services Export Version 9. http://www.ietf.org/rfc/rfc3954.txt.
[32] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better NetFlow for data centers," in *NSDI*, 2016, pp. 311–324.
[33] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "SketchVisor: Robust network measurement for software packet processing," in *ACM Special Interest Group on Data Communication*, 2017, pp. 113–126.
[34] https://sflow.org/.
[35] R. Hofstede, P. Čeleda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow monitoring explained: From packet capture to data analysis with NetFlow and IPFIX," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2037–2064, 2014.
[36] http://dpdk.org/doc/guides/sample_app_ug/performance_thread.html.
[37] https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt.
[38] M. Molina, S. Niccolini, and N. Duffield, "A comparative experimental study of hash functions applied to packet sampling," in *IEEE ITC*, 2005.
[39] G. Nassopulos, D. Rossi, F. Gringoli, L. Nava, M. Dusi, and P. M. S. del Rio, "Flow management at multi-Gbps: tradeoffs and lessons learned," in *IEEE TMA*, 2014.
[40] R. Jain and I. Chlamtac, "The P2 algorithm for dynamic calculation of quantiles and histograms without storing observations," *Communications of the ACM*, vol. 28, no. 10, pp. 1076–1085, 1985.
[41] https://github.com/ztz1989/p2-algorithm.
[42] https://fd.io/.
[43] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *ACM/IEEE ANCS*, 2015.
[44] M. Manesh, K. Argyraki, M. Dobrescu, N. Egi, K. Fall, G. Iannaccone, E. Kohler, and S. Ratnasamy, "Evaluating the suitability of server network cards for software routers," in *ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, 2010, p. 7.
[45] http://dpdk.org/doc/api/rte__thash_8h.html.
[46] http://burtleburtle.net/bob/hash/spooky.html.
[47] https://github.com/PeterScott/murmur3.
[48] A. Botta, A. Dainotti, and A. Pescapè, "Do you trust your software-based traffic generator?" *IEEE Communications Magazine*, vol. 48, no. 9, pp. 158–165, May 2010.
[49] https://perf.wiki.kernel.org/index.php/Main_Page.
[50] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe Performance for End Host Networking," in *Proceedings of the 2018 ACM Special Interest Group on Data Communication*, ser. SIGCOMM, 2018.

Tianzhu Zhang (M'14) obtained his Bachelor of Science degree in June 2012, from the Computer Science and Technology Department of Huazhong University of Science and Technology (HUST) in Wuhan, China. In October 2014, he obtained his master of science degree from Politecnico di Torino in Turin, Italy. From November 2014 till November 2017, he was a Ph.D candidate under the supervision of Prof. Paolo Giaccone. He also did an internship at Nokia Bell Labs, under the supervision of Dr. Massimo Gallo. From October 2017, he has been working as a research engineer at Telecom ParisTech, under the supervision of Prof. Dario Rossi. His main research interests are high-speed packet processing and topics related to NFV/SDN.

Leonardo Linguaglossa is a post-doctoral researcher at Telecom ParisTech (France). He received his master degree in telecommunication engineering at University of Catania (Italy) in 2012. He pursued a Ph.D. in Computer Networks in 2016 through a joint doctoral program with Alcatel-Lucent Bell Labs (nowadays Nokia), INRIA and University Paris 7. Leonardo's research interests focus on architecture, design and prototyping of systems for high-speed software packet processing, future Internet architecture and SDN.

Massimo Gallo is a member of technical staff at Nokia Bell Labs, Paris, France since May 2013. He received the Bachelor and the Master of Science degrees in Computer and Communication Networks from Politecnico di Torino in 2006 and 2008. He obtained the Ph.D. in Networks and computer science from Telecom ParisTech, Paris, France in 2012, performing his graduate research at Orange Labs, France Telecom, Paris, France. His main research interests are on the performance evaluation, simulation, design and experimentation on networked systems with particular focus on Programmable networks, Traffic generation, and Network Function Virtualization.

Paolo Giaccone (M'99, SM'16) received the Dr.Ing. and Ph.D. degrees in telecommunications engineering from the Politecnico di Torino, Italy, in 1998 and 2001, respectively. He is currently an Associate Professor in the Department of Electronics and Telecommunications, Politecnico di Torino. During the summer of 1998, he was with the High Speed Networks Research Group, Lucent Technology-Bell Labs, Holmdel, NJ. During 2000-2001 and in 2002 he was with the Information Systems Networking Lab, Electrical Engineering Dept., Stanford University, Stanford, CA. His main area of interest is the design of network control and optimization algorithms.

Dario Rossi (SM'13) is a Professor at Telecom ParisTech and Ecole Polytechnique, and is the holder of Cisco's Chair NewNet@Paris. He served on the board of several IEEE Transactions, and in the program committees of over 50 conferences including ACM ICN, ACM CoNEXT, ACM SIGCOMM and IEEE INFOCOM (Distinguished Member 2015, 2016, 2017 and 2019). He has co-authored 9 patents and over 150 papers, receiving 7 best paper awards, a Google Faculty Research Award (2015) and an IRTF Applied Network Research Prize (2016)