FlowMon-DPDK: Parsimonious per-flow software monitoring at line rate

Tianzhu Zhang[1,2], Leonardo Linguaglossa[1], Massimo Gallo[3], Paolo Giaccone[2], Dario Rossi[1]
[1]Telecom ParisTech, Paris, France – {tianzhu.zhang, linguaglossa, dario.rossi}@telecom-paristech.fr
[2]Politecnico di Torino, Turin, Italy – {tianzhu.zhang, paolo.giaccone}@polito.it
[3]Nokia Bell Labs, Nozay, France – massimo.gallo@nokia-bell-labs.com

***Abstract*—Testing experimental network devices requires deep performance analysis, which is usually performed with expensive, not flexible, hardware equipment. With the advent of high-speed packet I/O frameworks, general purpose equipments have narrowed the performance gap in respect of dedicated hardware and a variety of software-based solutions have emerged for handling traffic at very high speed. While the literature abounds with software traffic generators, existing monitoring solutions do not target worst-case scenarios (i.e., 64B packets at line rate) that are particularly relevant for stress-testing high-speed network functions, or occupy too many resources.**

In this paper we first analyse the design space for high-speed traffic monitoring that leads us to specific choices characterizing FlowMon-DPDK, a DPDK-based software traffic monitor that we release as an open source project. In a nutshell, FlowMon-DPDK provides tunable fine-grained statistics at both packet and flow levels. Experimental results demonstrate that our traffic monitor is able to provide per-flow statistics with 5-nines precision at high-speed (14.88 Mpps) using an exiguous amount of resources. Finally, we showcase FlowMon-DPDK usage by testing two open source prototypes for stateful flow-level end-host and in-network packet processing.

I. Introduction

Evaluating the performance of experimental devices and network applications requires intensive measurement campaigns on real prototypes, where several variables come into play. The procedure follows the guidelines provided by RFC 2544 on benchmarking network devices, as 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, typically the worst-case (e.g., a stream of 14.88 Mpps 64B-packets in case of a 10 Gbps interface). To measure its performance (maximum sustainable throughput, packet loss, etc.) the DUT in turn relays the received packets to a traffic monitor (RX). Similarly, in case of closed-loop experiments, the traffic monitor captures packets exchanged between transmit and receive sides of the DUT (i.e., a client-server application) and evaluates their performance.

The TX component presents two possible design approaches. The first one is leveraging expensive commercial equipments (i.e., hardware traffic generators), capable of generating line-rate traffic with high accuracy but low-to-none programmability. A second approach is employing a software traffic generator, which may provide less accurate generation, but a higher level of flexibility. A similar discussion applies for the RX component too. Hardware solutions can provide accurate measurements on a specific set of preset but monitoring non-default variables may be difficult 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 mis-computations. Additionally, RX resources can share the same hardware of the TX (or DUT in some case) for which *lightweight* operation is a very desirable property.

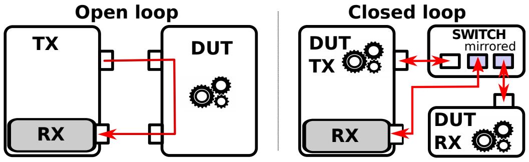


Fig. 1: Performance evaluation of a Device Under Test (DUT)

The last decade has witnessed a dramatic advancement of high speed I/O frameworks such as DPDK [1], PFQ [14], and netmap [2]. By means of kernel bypassing, batch-processing and poll-mode fetching, such frameworks provide general purpose hardware with the ability to capture and process packets in excess of 10 Gbps. This has resulted in the rise of software traffic generators (TX) [9], [18], capable of saturating a 10 Gbps link with minimum-size 64B packets by using one (or few) core(s) on commodity hardware. Despite the great availability of multiple choices on TX side, this is not yet the case for the RX one. Software monitoring tools either offer simplistic capabilities (e.g., per-packet operations only), or require a significant amount of resources to sustain processing of worst-case traffic and hence can hardly be co-located with TX or DUT.

In this paper, we present FlowMon-DPDK, a flexible traffic monitor based on DPDK that, beyond the typical *per-flow* statistics (e.g., flow size), is also able to perform more complex tasks, such as computing high order statistics (e.g., percentiles) of more involved metrics (e.g., per-flow interleave gap) while using the minimum amount of processing resources. We first revise existing traffic generator and monitoring literature (see Sec. II) and then outline our choices in the design space (Sec. III), which we experimentally verify (Sec. IV) by systematically benchmarking FlowMon-DPDK performance in a controlled setup (Sec. V). Our results show that, by carefully choosing the appropriate data structures, packet- and flow-level statistics incur negligible overhead (orders of magnitude smaller than available tools [3], [18]). Finally we also provide similar evaluations with publicly available traffic traces and showcase FlowMon-DPDK capabilities to monitor open- and closed-loop traffic of two open-source prototypes: ClickNF [4], [23], a modular stack for the composition of L2-L7 network functions, and FD.io's Vector Packet Processor (VPP) [5], a packet processing framework for software routers. We release FlowMon-DPDK as an open source project [6], which advances the state-of-the-art by providing fine-grained flow-level statistics at high-speed using the least possible amount of CPU resources.

II. Related work

Previous works on high-speed traffic generation (TX), processing (DUT) and monitoring (RX), relate to our contribution. Given our monitoring focus, we refer the reader to [13], [19] for a state of the art on TX and DUT, respectively. We purposely breakdown the available literature on traffic monitoring into *basic* vs *advanced*. FlowMon-DPDK sits in between of the two, as it attempts at doing operations pertaining to the latter class, while using as few resources as tools in the former.

A. Basic traffic monitoring

Traffic monitoring is bundled with TX tools such as MoonGen [18] and pktgen-DPDK [9], which not only support minimum-size packet generation at line-rate, but also offer basic packet-level measurement capabilities. In particular, MoonGen [18] is a high-speed traffic generator based on DPDK framework. In addition to generation, MoonGen takes advantage of hardware features for rate control and latency measurement. Since MoonGen scripts are based on 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. Alternatively, DPDK-Speedometer [3] is a packet-level traffic monitor written in C, and can be used to measure bandwidth consumption and throughput at line rate, making it a good comparison candidate for packet-level operations. These basic RX monitoring capabilities have traditionally sufficed for prototype design, especially since network function 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 (see [23] and references therein) started challenging the usefulness of packet-level monitors. As such, with respect to DPDK-Speedometer or MoonGen, FlowMon-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.

Finally, [15] 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 in a 10Gbps link with 64B packets (using at least 3 different RX cores). This optimized `pcap` can be even used with advanced monitoring tools such as Tstat [21], with slightly decreased performance (i.e., when using 3 Tstat instances and 128B packets the packet loss is a few percent points). On the contrary FlowMon-DPDK targets (i) minimum-size packets, (ii) aims to use a smaller number of CPU cores to perform not only packet-level measurements and (iii) keeps the packet loss ratio in the order of few parts per million. Note that, although there are already products able to achieve up to 100 Gbps rates, FlowMon-DPDK focuses on traffic monitoring with minimal resources. It is just a horizontal expansion of the required resources to switch from 10 Gbps to 100 Gbps.

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. At local level, which is the closest to this work, a number of sophisticated monitoring tools are reviewed in [30]. Among them, nTop [7] and DPDKStat [30] use the least amount of resources in terms of CPU cores, while being sophisticated tools designed for Internet traffic monitoring [21]. Whereas Deep Packet Inspection (DPI) and other advanced analytics can be deactivated, the TCP-based statistics are deeply entangled in the software and would require a complex code modification to be deactivated. For instance, according to [7] "using a dual core CPU, nProbe can be used for capturing packets at 1 Gbits with very little loss ($<1\%$)", whereas on similar hardware the simpler flow-level statistics tracked by FlowMon-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 $\approx$ 716 bytes range corresponding to $\approx$ 385 kpps. In contrast, FlowMon-DPDK achieves operation rates of about 3.7 Mpps per-core, *an order of magnitude* more than DPDKStat and *several orders of magnitude* with respect to Bro, Snort and Suricata [30].

As such, the capabilities of these advanced monitoring tools do not make them a good point 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, and thus the design choices. Alternatively to on-line monitoring, and in contrast to FlowMon-DPDK, some traffic monitors can capture and store packets for off-line analysis. In particular FlowScope [20] is capable of continuously capturing a subset of flows that can be further dumped into disk through predefined triggers. As FlowScope's goals are significantly different from ours, we do not consider it for direct comparison.

At network level, NetFlow [17] 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

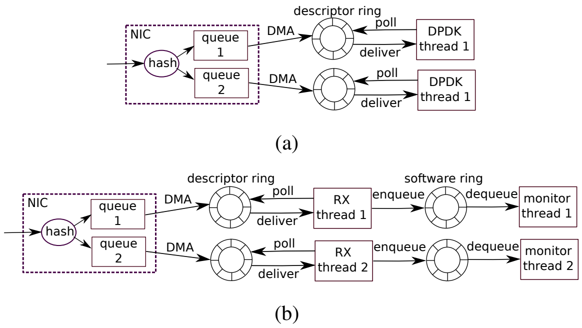


Fig. 2: Packet capture under (a) run-to-completion and (b) pipeline programming models

NetFlow. As such, to reduce the size of the data to be sent to the collector, FlowRadar [27] proposes to store each flow counter using a compact data structure based on counting bloom filters. To further reduce data size, SketchVisor [25] proposes to split the data collection path into regular and fast-path, which is used on load surges and which only performs updates locally at the switch for small portions of the heavy-hitter flows. sFlow [11] 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 faces the downside of inaccuracy due to sampling. Although NetFlow (and variants) perform operations similar to ours, they differ from FlowMon-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 to directly compare experimentally.

III. System design

Flow monitoring can be decomposed into four stages [24], namely *packet capture* to retrieve and pre-process packets when needed, *flow aggregation* to aggregate packets belonging to the same flow, *data collection* to store the collected statistics, and *data analysis* to provide the desired metrics characterizing the different flows[1]. In this section, we briefly review each of them and describe FlowMon-DPDK design according to this classification.

A. Packet capture

The first stage consists in retrieving packets from the line card, timestamping and forwarding them to the flow monitor engine. To capture and process all packets (i.e., without sampling) FlowMon-DPDK builds on top of DPDK, a fast packet I/O framework that enables user-space packet reception at line rate by means of kernel bypass, batch, multi-queue, and multi-thread processing. To optimize FlowMon-DPDK performance, in the following we consider different design choices such as multi-threading and DPDK programming models.

Programming models: DPDK provides two alternative programming models. In *run-to-completion*, each packet is retrieved and processed by the same thread. As shown in Fig. 2-(a), each thread keeps polling the receive descriptor ring and fetching a batch of packets into user space for further processing. Instead, in the *pipeline* model, the workload is distributed to a group of threads. As shown in Fig. 2-(b), FlowMon-DPDK pipeline model has two different types of threads: rx- and monitor-thread. The former keeps polling the descriptor rings to retrieve packets, and the latter fetches retrieved packets from the rx-threads through the corresponding software ring for further processing (i.e., collect flow statistics). Although such programming models are consolidated, it is uncertain whose performance is superior in our case. In this paper, we test both of them and evaluate their performance under packet- and flow-level processing, which hopefully provides guidelines with a wider applicability than the boundaries of our work.

Multi-threading models: DPDK provides its own multi-threading model, namely *lthread* [8], as an alternative to the standard POSIX *pthread* library. Unlike pthread, lthread embraces cooperative scheduling in which threads cannot be preempted, and they need to periodically yield their execution. Thus lthread yields lower scheduling overhead, contention avoidance and per-thread local storage. In our context where threads perform relatively simple operations (e.g., few table and memory accesses), lthread seems a lightweight appealing alternative: we implement both models in FlowMon-DPDK and perform a selection by benchmarking their performance.

B. Flow aggregation

After being captured, packets are aggregated into flows by means of a flow identifier, typically the TCP-UDP/IP 5-tuple and stored in a hash table. While it is desirable for hash functions to have good entropy properties [28], computing the hash has a non-marginal penalty in performance [29], for which simple functions (sum or xor) are generally used [21]. We argue that already accessing the 5-tuple elements incurs a non-negligible overhead. As the NIC already computes some hash (i.e., Toeplitz) over the 5-tuple when Receive Side Scaling (RSS) is enabled, we further argue that is preferable to access and reuse this single pre-computed 32-bit hash value, as opposite to access multiple (albeit contiguous) memory portions over which to recompute a hash function.

Once the flow hash is available, it is used to index a table storing the per-flow statistics. Clearly, the most naïve solution of directly accessing an array with 2^{32} entries requires significant amount of memory, especially given that the structure should be preallocated with counters for the task at hand (e.g., flow size counting, flow interleaving gap, etc.). We thus consider three approaches, that are (pedagogically) reported in Fig. 3.

Double hash: A data structure with good properties is the *double hash*, as in Fig. 3-(a). The table contains only 2^{16} buckets, each of which hosts two entries. Flows are indexed using only a portion of the RSS hash (e.g., the lowest bits) while the remaining RSS hash portion (16 high bits) is used to

[1] Note that all stages of FlowMon-DPDK are performed locally on the server, thus the last two stages are not performed through remote flow collectors such as the case for NetFlow.

discriminate flows within a bucket. The structure is efficient, as it is cache aligned, but not accurate in presence of several concurrent flows as collisions are poorly handled. Indeed, due to the birthday paradox, on average $1.25\sqrt{2 \cdot 2^{16}} \approx 450$ concurrent flows can be tracked without collisions.

Linked-list hash: To resolve hash conflicts, one option is to allow multiple entries in the same bucket through a linked list, as in Fig. 3-(b). This approach avoids the saturation of a bucket and allows a correct count for all the flows, but suffers from the overhead of dynamic memory allocation and of non-contiguous memory accesses, as well as the time for the linear search 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. At the same time the payoff of these advanced structures is low in practice since the overhead of tree management offset most of the gain [29].

Combined hash: To simultaneously retain performance and correctness, we combine the previous two data structures, by appending the double hash with a linked list in each bucket, so as to resolve *all* hash conflicts, as in Fig. 3-(c). In this way, double hash miscounting is removed, while at the same time we avoid the higher cache miss rate due to non-contiguous memory space of the flow buckets in the linked-list hash case.

C. Data collection

The 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 nTop [7], DPKDStat [30], Netflow [17] or FlowScope [20], FlowMon-DPDK data collection is very simple (i.e., post-mortem reports of per-flow statistics or at periodic configurable intervals), exploits volatile memory and does not offer advanced features for export. As such, a thorough accounting of data collection in FlowMon-DPDK is outside the scope of this paper.

D. Data analysis

FlowMon-DPDK supports both *per-packet* and *per-flow* analysis. At packet-level, it is capable of hardware (HW) and software (SW) packet counting. Per-packet HW counting queries the NIC registers and gets the packet statistics, including transmitted/received packets/bytes, as well as total dropped ones due to the saturation of internal queues. Per-packet SW counting simply maintains per-queue packet counters and updates them upon arrivals of traffic batches: thus, statistics can be derived using a pure SW approach – which allows to assess the overhead of SW for very simple tasks.

Per-flow analytics are only available in software due to HW limitations, and make 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, which allows to appreciate the burstiness of a given flow. In particular, for a given flow i, we define the inter-leaving gap I_i as the number of packets received between two consecutive packet arrivals of flow i, as shown in Fig. 4. To stress test FlowMon-DPDK, we monitor the per-flow 99th percentiles of the inter-leaving gap implemented using the well-known PSquare algorithm [26], which requires only to keep track (and update at each packet reception) of a constant amount of state (in the order of a small tens of memory accesses and some floating point operations). While significantly simpler than the average task performed by advanced monitoring tools such as DPDKStat [30], this task already represents a stressful scenario for RX tools that aim at using a very limited amount of resource.

TABLE I: Details of the traces in the experiments

Trace	Rate (Mpps)	Avg pkt. size (Bytes)	# packets	# flows	Type
1	2.07	583	7.24×10^8	5514	ISP
2	11.12	92	3.88×10^9	17006	malware

IV. Testbed environment

1) Hardware: Our testbed consists of two commodity servers, running Linux 4.4.0+ distributions and equipped with Intel(R) Xeon(R) 2.60GHz CPUs, 32k/256k/25600k L1-3 caches, 64G RAM, and Intel(R) 82599ES 10-Gbps NICs[2]. Unless otherwise stated, there is no DUT, the two servers (TX and RX) are directly connected, and Ethernet flow control is disabled. This allows us to precisely measure the amount of loss due to the RX, without incurring loss due to DUT. According to [5], we configure the servers to reserve cores for the exclusive usage of FlowMon-DPDK so as not to degrade the performance. Then the CPU frequency scaling governors are set to "performance" for all the active cores to maximize the processing speed. We set 1GB Huge Pages to enable efficient page lookup in Linux.

2) Software: For our tests structured as in Fig. 1, we use MoonGen as TX. We use FlowMon-DPDK (DPDK 17.05) as RX. As main performance metric we compute the Packet Drop Ratio (PDR), i.e., the average drop probability – we stress 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 due to RX, in Sec. VI we additionally consider the throughput with ClickNF [23] and VPP [5] as DUT, as terms of comparison.

3) Traffic workload: For the initial experiments, we adopt open-loop traffic. MoonGen (TX) (commit 31AF6E6 on 31/10/2017) is used to generate synthetic traffic and trace-driven traffic. Under synthetic traffic (Secs. V, VI-A), all packets are 64B and injected at 10 Gbps (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 to a Zipf law (with $\alpha = 1$) across a set of 2^{16} flows, corresponding to an hash load factor 1.0. Under trace-driven traffic (Sec. VI-A), MoonGen replays the traces described in

[2] Note that the NICs we use are equipped with a relatively old chip but we believe similar results hold with newer NICs.

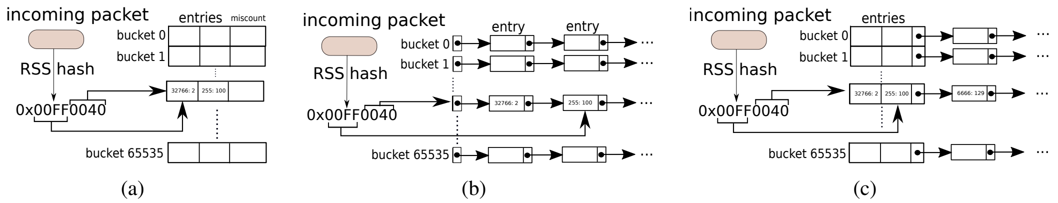


Fig. 3: Flow table implemented with (a) double hash, (b) linked-list hash, (c) combined hash.

.

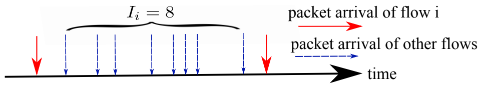


Fig. 4: Example of per-flow inter-leaving gap evaluation

Table I. To measure the loss rate with high precision (reported in a 10^{-6} scale, i.e., part-per-million), each experiment lasts for 6 minutes (over 5 billion packets), and all the graphs report the 95% confidence intervals averaged across 50 repetitions of the same test. In the case of trace-driven traffic, the trace is repeated in a loop to reach the target duration.

For the final experiments, in Sec. VI-B we use closed-loop traffic, generated with a fast TCP server/client implementation. In particular we use ClickNF [4], [23] a modular network L2-L7 stack for end-host and middlebox functions. A given pool of ClickNF clients connects to the ClickNF server using random input ports in the range [1-65535], i.e., 65k different flows as in the previous scenarios. Once the connection is established, the 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 open-loop traffic, in Sec. VI-B we consider as DUT a software router using the stack provided by VPP [5]. In particular, we consider the case in which the VPP router employs a per-flow scheduling algorithm to enforce bandwidth fairness among flows [12]. 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 kth flow is k times slower than the first one (i.e., the fastest).

V. Software tuning

In this section, we carefully validate our system by comparing the different alternatives. We first conduct 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 "default" values yielding the best performance that are used unless otherwise specified.

A. Packet capture stage

Batch size: The batch size is the maximal number of packets fetched through one poll-mode fetch in DPDK. Similar to [22], we evaluate the impact of batch size ranging from 64 to 1024 packets: the tradeoff is that small batches cannot properly handle RX bursts and lead to losses in the NIC, whereas too long batches may take longer to process and yield to losses in the RX. Fig. 5-(a) shows that losses are minimal for 256 packets batches – an operational point at which the DPDK Direct Memory Access (DMA) operations are likely optimized and that we set as default for the following experiments.

TABLE II: Optimal running parameters for FlowMon-DPDK.

Parameter	Value
Flow data structure	combined (double hash + linked list)
Batch size	256 packets
Number of RSS queues	2
Number of mempools	one for each RSS queue
Hyper-threading	disabled
Programming model	pipeline with pthread
Flow analytics	per-flow packet counting

Number of HW queues: By enabling RSS, the load of incoming traffic is distributed into multiple queues, that are associated with different threads (or cores). While increasing the number of RSS queues is beneficial for load balancing purposes, increasing the number of RSS queues can lead to contention on the PCI bus. Fig. 5-(b) reports the PDR for increasing number of RSS queues, from which we infer that 2 queues are optimal to handle worst-case traffic on our commodity hardware (we thus fix RSS=2 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 issue does not appear in tools performing more complex analytics [30] on bigger packets, where CPU is a more stringent bottleneck.

Per-queue memory pool: DPDK reserves pools of packet descriptors to associate incoming packets from the NIC, thus avoiding the overhead of dynamic 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 mempools in the following.

Programming models: Finally, we implement two flavors of FlowMon-DPDK. The first one adopts the run-to-completion

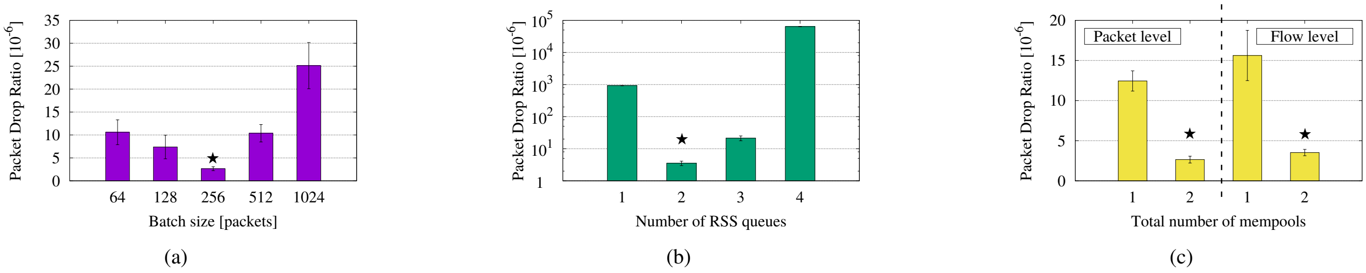


Fig. 5: Impact of (a) batch size (b) number of RSS queues, and (c) of mempools. Default parameters are denoted with $\star$.

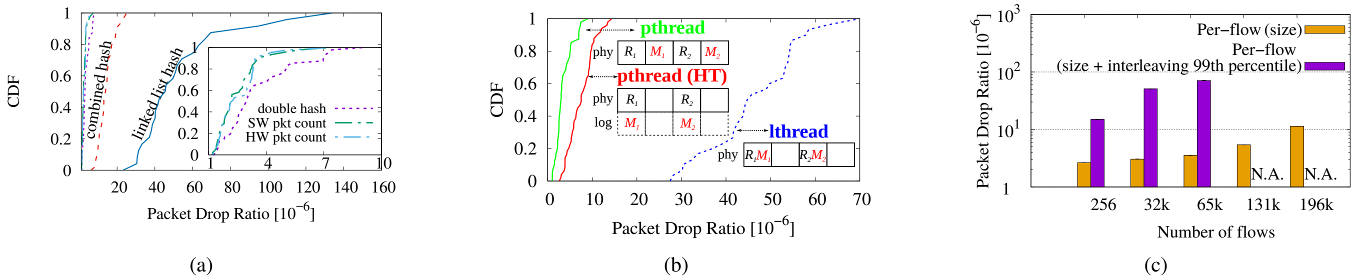


Fig. 6: CDF of packet drop ratio for (a) different data structures (b) configurations for the combined structure, and (c) performance comparison for simple (per-flow packet counting) and complex (per-flow 99th percentile of the inter-leaving gap) statistics with increasing number of flows(c).

TABLE III: Comparison of the packet drop ratio for different programming models. A $\star$ sign denotes default parameters.

Parameter	Average PDR	95% C.I.
Run-to-completion	4.8×10^{-6}	5.7×10^{-7}
Pipeline with pthread $\star$	3.5×10^{-6}	5.6×10^{-7}

model while the other is based on the pipeline model. As shown in Table III-(e), the pipeline model outperforms the run-to-completion one (for the sake of space, we report only results for pthread). In 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 big buffer that absorbs the processing delays of packet monitoring threads, thus achieving maximum throughput.

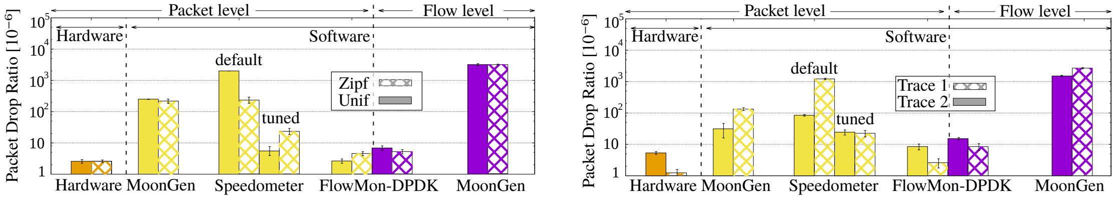
B. Flow aggregation stage

Data structure: Fig. 6-(a) compares FlowMon-DPDK performance with different data structures and shows HW and SW packet counting as a reference. Per-flow SW counting performance is significantly affected by the data structure. In particular, loss of double hash is comparable with that of SW packet counting, though double hash can present miscounting and should be avoided in case of high load on the hash table. Conversely, single hash with linked list 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 rates. Finally, combined hash sits at the intermediate point in the performance tradeoff, achieving precise counting with a limited overhead. Per-packet SW and HW losses are very similar, and provide a lower bound for the PDR of flow-level measurement, settling around a measurable $2 \cdot 10^{-6}$ on average. Note that packet losses with HW, SW packet counting are negligible[3] and mainly due to software generation inaccuracy [16], [19].

Implementation choices: Flow aggregation stage performance depends on implementation details, such as the offloading of hash computation to the NIC (which spares useless memory accesses and saves useful CPU cycles) as well as the selection of libraries for thread implementation. In particular, with FlowMon-DPDK we verified that computing the hash in software[4] results in a PDR of about 30%, translating to a maximum RX rate of 10.5 Mpps. Considering the double-hash implementation (for the sake of simplicity) and limitedly focusing on the pipeline model (similar results holds for run-to-completion model) we contrast the lthread vs pthread implementations. The pipeline model consists of two threads, a receiver (R) and a monitoring (M) thread for each RSS queue, i.e., 4 distinct threads (R_1, R_2, M_1, M_2) for the total 2 RSS queues, The allocation of threads to cores is illustrated in Fig. 6-(b). Specifically, the four cells in the first row represent 4 physical cores, while those in the second row the virtual cores, enabled by hyper-threading. In case of lthread, cooperative multi-threading is implemented within the application itself: R and M threads coexist in the same core and the application manages their execution. In case of pthread, threads can be placed onto different physical cores, or on two logical cores of the same physical one.

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

[4] Using the optimized DPDK `rte_softrss_be()` function.



(a) (b)

Fig. 7: Traffic monitoring: performance under (a) Synthetic traffic with Uniform and Zipf patterns (b) Realistic traffic traces.

We see that pthread in hyper-threading significantly decreases the packet losses with respect to lthreads; 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 [30]. 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 with a cache-friendly memory structure, may diminish the usefulness of hyper-threading (similar issues are also observed in optimized software stack VPP [5] with high instruction-per-clock efficiency).

C. Data analysis stage

Finally, we analyze two types of analytics: per-flow packet counting (only per-flow packet and byte counters are recorded) and the per-flow 99th percentile of the inter-leaving gap (which requires access and manipulation of more states).

Results are shown in Fig. 6-(c), where the simple and complex analytics are executed over 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 seen 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 a parsimonious amount of resource. Note that for higher hash load, then the losses are too high for the targeted scenarios.

VI. Experimental results

We test FlowMon-DPDK in different scenarios. Specifically, Sec. VI-A contrasts the performance of FlowMon-DPDK to that of other tools under synthetic as well as trace-driven traffic. Next, Sec. VI-B 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).

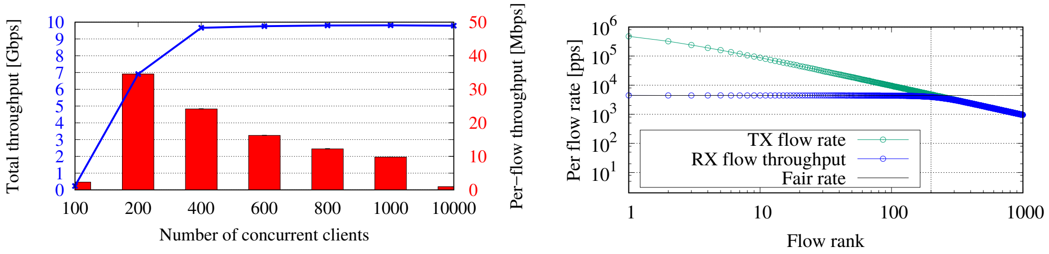
A. Traffic monitor precision

Synthetic traffic: Fig.7-(a) shows the performance for HW packet counting (left, orange) vs SW packet counting (center, yellow) vs SW per-flow monitoring (right, violet). Experiments under uniform and Zipf distribution of the flows are reported with solid color and hashed pattern, respectively. HW packet counting is reported as a reference. Packet-level counting performance are reported for HW, MoonGen, Speedometer, and FlowMon-DPDK. For MoonGen, the monitoring features are all built upon its underlying framework named libmoon [10]. For Speedometer, we additionally report "default" and "tuned" settings. In the "default" ones, Speedometer is used unmodified and reports two orders of magnitude more losses than HW. In the "tuned" ones, Speedometer has the same FlowMon-DPDK tuning adopted in Sec. V and results in one order of magnitude more losses than HW, similarly to MoonGen. We also note that FlowMon-DPDK exhibits an advantage (albeit slight) over Speedometer and MoonGen possibly because they perform more operations, although we did not dig further as we focused our study mainly on flow-level performance.

Finally, we consider per-flow counting, contrasting FlowMon-DPDK with a 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 (so 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 increases losses by one additional order of magnitude. Second, activating per-flow counting in FlowMon-DPDK has only a negligible effect as far as losses are concerned.

Trace-driven experiments: We test the performance under the traffic traces of Table I. Realistic traffic may exhibit temporal correlation that the independent traffic models used in experiments with synthetic traffic do not capture. It follows that performance may differ with respect to the synthetic traffic case (e.g., temporal correlation may induce cache hits). Clearly, different temporal traffic patterns and spatial flow distribution may affect the results, which are thus to be interpreted in qualitative sense.

In particular, from the experiments reported in Fig. 7-(b), one can expect from FlowMon-DPDK a loss rate in the order of few parts-per-million $[10^{-6}, 10^{-5}]$. Part of this loss rate is physiologic in the capture process (coherently with HW packet counting) and part of it is due to the speed of operation in packet processing in software. From these DUT-less experiments, we can thus conclude that FlowMon-DPDK is able to measure per-flow statistics with 5-nines precision and that any measured per-flow loss rate above the 10^{-5} threshold has therefore to be imputed to the DUT.



(a)

(b)

Fig. 8: DUT testing: (a) closed-loop transport function in ClickNF and (b) open-loop scheduling function in VPP.

B. Traffic monitor usage

ClickNF transport: Fig. 8-(a) shows the per-flow (bars) and total throughput (line) measured in the ClickNF client-server scenario, as described in Sec. IV. 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 packet that need to be transferred. Indeed it is known that, when processing small batches, DPDK exhibits poor performance mainly due to the overhead of transferring over 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 the line rate and FlowMon-DPDK provides a handy complement to the ClickNF log, allowing to check fairness and efficiency of the TCP server implementation.

VPP scheduling: Fig. 8-(b) shows TX and the VPP traffic measured with FlowMon-DPDKin the scenario described in Sec. IV. It can be noticed that the first 200 most aggressive flows observe a decrease in their rate because they exceed their fair share of the link, while the output rate of the remaining flows matches their input rate, since their rate is lower than their fair share. Note that to reliably monitor low-rate flows, and thus check the correctness of the implementation, FlowMon-DPDK reliability is of uttermost importance.

VII. Conclusions

In this paper, we design, tune and experiment FlowMon-DPDK, a tool capable of providing fine-grained per-packet and per-flow statistics at line rate, using a minimal amount of resource. The tool leverages RSS hash computation to offload software operations, and employs a careful design where flow-tables are aligned with cache line boundaries. Thanks to its careful design, FlowMon-DPDK outperforms state-of-the-art alternative solutions and provides the researchers a precise, inexpensive tool for monitoring the performance of their high speed prototypes. We make the tool available at [6].

Acknowledgments

This work benefited from support of NewNet@Paris, Cisco's Chair "Networks for the Future" at Telecom ParisTech.

References

[1] http://dpdk.org/.
[2] http://info.iet.unipi.it/~luigi/netmap/.
[3] https://github.com/hpcn-uam/.
[4] https://github.com/nokia/ClickNF.
[5] https://wiki.fd.io/view/VPP.
[6] https://github.com/ztz1989/FlowMon-DPDK.
[7] https://www.ntop.org/.
[8] http://dpdk.org/doc/guides/sample_app_ug/performance_thread.html.
[9] Intel Pktgen-DPDK. https://github.com/pktgen/Pktgen-DPDK.
[10] libmoon. https://github.com/libmoon/libmoon.
[11] sFlow. https://sflow.org/.
[12] V. Addanki, L. Linguaglossa, et al. Controlling software router resource sharing by fair packet dropping. In *IFIP Networking*. 2018.
[13] T. Barbette, C. Soldani, et al. Fast userspace packet processing. In *ACM/IEEE ANCS*. 2015.
[14] N. Bonelli, A. Di Pietro, et al. On multi–gigabit packet capturing with multi–core commodity hardware. In *PAM*. Springer, 2012.
[15] N. Bonelli, S. Giordano, et al. Enabling packet fan-out in the libpcap library for parallel traffic processing. In *IEEE TMA*. 2017.
[16] A. Botta, A. Dainotti, et al. Do you trust your software-based traffic generator? *IEEE Communications Magazine*, 48(9):158, 2010.
[17] B. Claise, G. Sadasivan, et al. RFC3954 Cisco Systems NetFlow Services Export Version 9. http://www.ietf.org/rfc/rfc3954.txt, 2004.
[18] P. Emmerich, S. Gallenmüller, et al. Moongen: A scriptable high-speed packet generator. In *ACM Conference on Internet Measurement Conference*, pages 275–287. 2015.
[19] P. Emmerich, S. Gallenmüller, et al. Mind the gap: A comparison of software packet generators. In *ACM/IEEE ANCS*. 2017.
[20] P. Emmerich, M. Pudelko, et al. FlowScope: Efficient packet capture and storage in 100 Gbit/s networks. In *International IFIP TC6 Networking Conference*. 2017.
[21] A. Finamore, M. Mellia, et al. Experiences of Internet traffic monitoring with Tstat. *IEEE Network Magazine*, 25(3):8, 2011.
[22] S. Gallenmüller, P. Emmerich, et al. Comparison of frameworks for high-performance packet IO. In *ACM/IEEE ANCS*. 2015.
[23] M. Gallo and R. Laufer. clicknf: a modular stack for custom network functions.
[24] R. Hofstede, P. Čeleda, et al. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials*, 16(4):2037, 2014.
[25] Q. Huang, X. Jin, et al. Sketchvisor: Robust network measurement for software packet processing. In *Conference of the ACM Special Interest Group on Data Communication*, pages 113–126. 2017.
[26] R. Jain and I. Chlamtac. The P2 algorithm for dynamic calculation of quantiles and histograms without storing observations. *Communications of the ACM*, 28(10):1076, 1985.
[27] Y. Li, R. Miao, et al. Flowradar: A better netflow for data centers. In *NSDI*, pages 311–324. 2016.
[28] M. Molina, S. Niccolini, et al. A comparative experimental study of hash functions applied to packet sampling. In *IEEE ITC*. 2005.
[29] G. Nassopulos, D. Rossi, et al. Flow management at multi-Gbps: tradeoffs and lessons learned. In *IEEE TMA*. 2014.
[30] M. Trevisan, A. Finamore, et al. Traffic analysis with off-the-shelf hardware: Challenges and lessons learned. *IEEE Communications Magazine*, 55(3):163, 2017.