Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned

Martino Trevisan[†], Alessandro Finamore[‡], Marco Mellia[†], Maurizio Munafò[†], Dario Rossi[⋆]
[†]Politecnico di Torino, [‡]Telefonica Research, [⋆]Telecom ParisTech,
{martino.trevisan, mellia, munafo}@polito.it
dario.rossi@telecom-paristech.fr
alessandro.finamore@telefonica.com

Abstract— **In recent years, the progress in both hardware and software enabled user-space applications to capture packets at 10 Gbit/s line rate. However, processing packets at such rates with software running on Commercial Off-The-Shelf (COTS) hardware is still far from being trivial. In the literature, this challenge has been extensively studied for Network Intrusion Detection Systems (NIDS), where operations are per-packet and easier to parallelize also thanks to hardware acceleration. Conversely, the scalability of Statistical Traffic Analyzers (STA) is intrinsically more complex as it implies tracking per-flow state to collect statistics. This challenge received less attention so far, and it is the focus of this work.**

We discuss the design choices to enable a STA to collects hundreds of per-flow metrics at a multi 10 Gbit/s line rate. We leverage a handful of hardware advancements proposed over the last years (e.g., RSS queues, NUMA architecture), and we provide insights on the trade-offs they imply when combined with state of the art packet capture libraries and multi-process paradigm. We outline the principles to achieve an optimized STA, and we apply them to engineer DPDKStat, a solution combining the Intel DPDK framework with the traffic analyzer Tstat. Using traces collected from real networks, we demonstrate that DPDKStat achieves 40 Gbit/s of aggregated rate with a single COTS PC.

I. Introduction

The last years have witnessed a growing interest towards solutions for Internet traffic processing. The engineering of such systems is a far from trivial challenge. In fact, if Internet services are becoming more and more complex and require more processing power to monitor them, at the same time the Moore law scales at a slower pace compared to the annual bandwidth consumption rate. Traffic monitoring requires to acquire, move, and process *packets*, while maintaining their logical organization in *flows*. These are daunting tasks to tackle at 10 Gbit/s line rate or more, where each packet lasts few tens of nanosecond. Running thus a software monitor on Common-Off-The-Shelf (COTS) hardware requires a lot of ingenuity.

The advent of both open source and proprietary packet acquisition libraries and ad-hoc hardware solutions alleviated the problem of the mere packets acquisition. These solutions indeed enable to achieve multi 10 Gbit/s line rate processing thanks to *zero-copy*, i.e., packets are moved via Direct Memory Access (DMA) by the Network Interface Controller (NIC) directly into user-space. The challenge then becomes how to speed up the extraction of valuable information from such a deluge of data. Software developers have explored multi-core CPUs, Graphical Processing Units (GPUs), Network Processing Units (NPUs), and FPGA architectures. This is testified by seminal [1] and more recent works [2], [3], [4] successfully scaling and optimizing multi-core Network Intrusion Detection Systems (NIDS), where a large set of rules have to be checked on a per-packet base. Fewer efforts have been instead devoted in the area of Statistical Traffic Analyzers (STAs) which instead aim to collect both basic statistics (e.g., TCP RTT or congestion events) and more articulated indexes (e.g., performance for video streaming and Webpage load time). STAs normally imply keeping per-flow state, hence they are inherently more difficult to scale than NIDS.

In this work, we report on our experience in designing and engineering DPDKStat, a system combining the Intel DPDK framework for packet acquisition, and the traffic analyzer Tstat [5], an STA offering a large number of per-flow metrics monitored *on-the-fly*.[1] We do not aim to present another fancy traffic monitoring tool. Conversely, we discuss *system bottlenecks* and *design principles* to overcome them. Moreover, we evaluate DPDKStat using real traces from 2 different scenarios ($\approx$ 20,000 ADSL residential customers of a major European ISP, and $\approx$ 10,000 users of a university campus network) using COTS solutions costing less than 4,000 UDS showing. Overall, DPDKStat achieves 40 Gbit/s thanks to a careful engineering of the trade-off behind packet acquisition, multi-process paradigm, and NUMA (Non Uniform Memory Access) architectures.

Summarizing, our major contributions are:

- We dissect different design choices, and evaluate them with traces that capture workloads representative of real scenarios.
- We investigate packets acquisition policies that guarantee consistent per-flow load balancing, limit timestamp errors, and avoid packets reordering and losses.
- We quantify benefits of periodic packets acquisition via `SCHED_DEADLINE` (+85%), hyper-threading (+20-30%), and load balancing across CPUs (+10%).

We make available to the community both DPDKStat and the traffic generator used in our testbed.[2] More details about DPDKStat are also available in [6].

[1]TCP and UDP traffic is processed at a per-packet base without reassembling IP fragments, nor rebuilding the actual content transmitted.
[2]http://tstat.polito.it/viewvc/software/tstat/branches/tstat-dpdk/

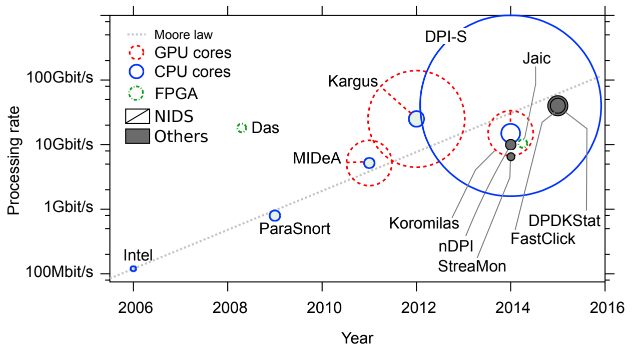


Fig. 1: Synoptic of related works. Circles are centered on the year and processing rate. Radius size is a logarithmic scaling of the number of cores employed by the system.

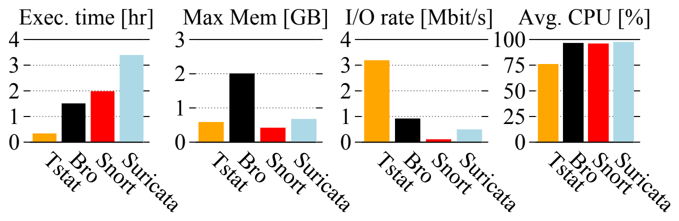


Fig. 2: STA and NIDS performance comparison (1-core, all tools with default configuration).

II. 10 Years of High Speed Traffic Processing

Both academia and industry has invested a large effort in designing efficient high speed Internet traffic processing systems. Since seminal work capable to cope with only few hundreds Mbit/s, different solutions passed the 10 Gbit/s "barrier". This is mostly thanks to the advent of advanced packet capture libraries (compared and benchmarked in [7]) which solved the first engineering challenge: efficiently transfer packets from the NIC to the main memory. Some works also address the problem to efficiently store packets on disks using COTS for later processing [8]. The challenge than becomes how to quickly process packets in user space, and this is usually achieved using parallelization and multi-cores technologies.

For the sake of illustration, we represent in Fig. 1 the most important solutions as circles centered at $(rate, year)$ with a radius proportional to the number of cores used. A straight line (in semi-log scale) represents Moore's law exponential increase of raw processing rate, doubling every year from the initial starting point of 100 Mbit/s. Comparison with old systems such as Intel[3] or ParaSnort is only anecdotal. Specifically, in 2015, the processing rate exhibits a speedup close to 2^{10} (2^6) with respect to the 2006 Intel system (2009 ParaSnort), well matching Moore expectations.

Notice that most of the works in Fig. 1 focus on NIDS (empty circles), i.e., they are Bro or Suricata based solutions [9]. These tools are designed to trigger alarms when a packet matches signatures from a predefined dictionary, but they report little statistics about the traffic activity. Thus, they work on a *per-packet* base or using simple state machines, and they are easily amenable to parallelization. However, since pattern matching is costly (e.g., a core can cope with only ~100 Mbit/s), NIDS scalability is achieved with a large number of GPU cores as in the case of MIDeA and Kargus [2], with NPUs as in Koromilas [4] and DPI-S [3], or finally with FPGAs as in Das [10] and Jaic [11]. Fig.1 also includes solutions that, despite not being STA, are not pure NIDS either. Specifically, StreaMon [12] is a SDN traffic monitoring framework, FastClick [7] is an advanced software router based on Click, while nDPI [13] is a pure traffic classifier derived from OpenDPI.

To the best of our knowledge, less effort has been devoted to study scalability issues for STAs (filled circles in Fig. 1). Such tools comprise a smaller, yet more varied, set of functions intrinsically more difficult to parallelize than NIDS. In fact, STAs functions share *per-flow* state, leading to a more pipelined analysis workflow than for NIDS. To culprit these tool classes, Fig. 2 compares processing time, maximum memory, I/O rate, and average CPU utilization when running Tstat (a STA) and three NIDs (Bro, Snort, and Suricata) on the same trace, using the same server, with tools in default configuration. Tstat is faster than the three other tools, but consumes I/O since it tracks hundreds of per-flow metrics. Notice also how, despite tracking per-flow states, Tstat it is lighter than Bro since it does not reassemble IP/TCP packets. This experiment however considers the usage of a single core which is insufficient to achieve multi 10 Gbit/s without parallelization. In the remainder of this work, we specifically dissect the design choices and the lesson learned to achieve such goal.

III. Design Principles

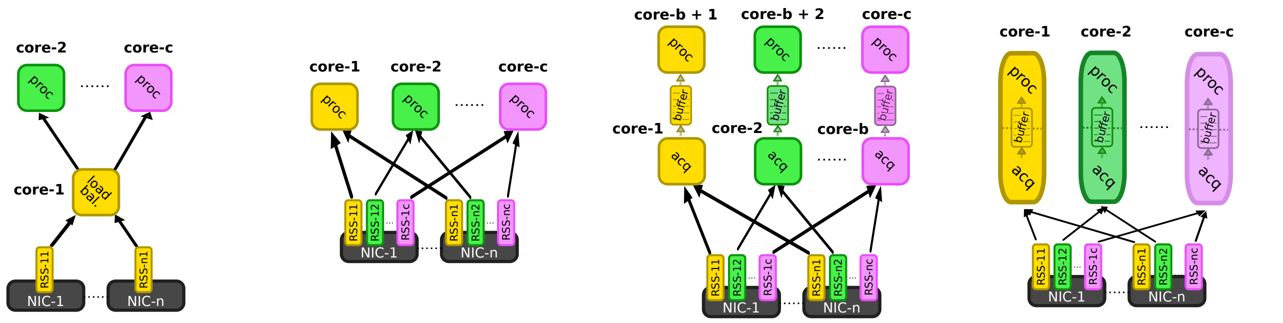
We assume that the STA runs on a COTS hardware monitoring 10 Gbit/s links. We assume the monitoring system to be equipped with n NICs, and c CPU cores. Notice that two interfaces are required for each single full-duplex link. To cope with the load, the STA needs to balance the traffic among different processing engines that are bound to different CPU cores. Fig. 3 shows the different design choices to be considered.

A. Packet acquisition and per-flow load-balancing

Goal. Several solutions have been proposed to provide efficient packet acquisition on COTS hardware. They all solve the problem of efficiently moving packets from the NICs to user-space [7], [8]. However, to compute per-flow statistics, we need to correlate packets received irrespective of the NIC where the packets are observed. Hence, the packet acquisition library needs to offer a *flow-preserving load balancing function* for correct traffic processing. This offers also the appealing opportunity to split the traffic among the c CPUs. The primary goal is to avoid costly synchronization primitives.

Proposal. A first option is to use *load balancing in software*

[3]http://courses.csail.mit.edu/6.846/handouts/H11-packet-processing-white-paper.pdf



(a) Software load balancer. (b) Direct RSS queues access. (c) Buffered access to RSS queues with dedicated core. (d) Buffered access to RSS queues with shared core.

Fig. 3: System architecture: the order is from the simplest one (left) to the most evolved and performing (right)

(Fig. 3a). This is offered by solutions such as PF_RING ZC[4] where custom per-packet load balancing can be coded and applied on the aggregate traffic received from the so called "DNA cluster", i.e., a group of NICs. In this case, all packets received from the NICs are passed to the DNA cluster process, which (i) timestamps and (ii) forwards them to the correct processing engine. Unfortunately, this solution does not scale as the software load balancer quickly becomes the bottleneck, and it is non optimal in multi-CPU scenarios where the same packet should be moved across the *NUMA* nodes of the system.

Modern NICs offer *load balancing in hardware*, e.g., via the Intel Receiver Side Scaling (RSS) queues. Consistent per-flow load balancing is possible with specific hashing functions [14] offloaded to the NIC. This results in a system where packets are stored into different RSS queues to which the STA has direct access. In this scenario, the number of RSS queues is equal to the number of CPU cores (Fig. 3b).

Offloading functionalities to hardware presents clear benefits, but the RSS technology suffers from some limitations. For instance, the load-balancing is performed only on IP packets encapsulated directly over Ethernet, excluding other Layer2 or tunneling protocols (MPLS, GRE tunnels, etc.). RSS queues are also a scarce resource (currently, at maximum 16 for each NIC). More important, they require careful tuning to properly timestamp packets (see Sec. V-A).

B. Absorbing traffic and processing jitters

Goal. Packet processing time is not constant. Traffic processing tools need to be engineered to minimize the *average* packet processing time. However, unexpected (large) processing delays typically occur due to slow I/O operations, periodic data structure optimization, critical packet composition, etc. These delays can lead to losses in the RSS queues since they can only store up to 4096 packets, i.e., few tens of microseconds at 10 Gbit/s. Similarly, unexpected or unbalanced traffic bursts can lead to losses too. Packet acquisition libraries already implement circular buffers to absorb such jitters. Yet, those are in the range of 1 MB and can only absorb less than one millisecond worth of traffic at 10 Gbit/s, i.e., this is a new problem specific for very high speed monitoring.

Proposal. Our solution is to decouple each analysis module using a *large buffer* (Fig. 3c). For instance, 1 GB is sufficient to store approximately 1 second at 10 Gbit/s. This requires a two threads system: (i) the *acquisition* thread extracts packets from the RSS queues, timestamps and enqueues them to the buffer tail; (ii) the *processing* thread dequeues packets from the buffer head and processes them. Normally such design choice would lead to expensive process synchronization. Fortunately, lock-free shared buffer data structures using state of the art zero-copy data acquisition are available. The presence of acquisition and processing threads complicates the CPU resource allocation. In fact, the RSS queues access is time critical so it should be operated on a dedicated core, while the processing is bound to a separate core. In summary, the design follows an "hybrid" approach: (i) different independent processes are attached to (a group of) RSS queues, but (ii) each process has separate threads managing acquisition and processing independently.

C. Efficient sharing of CPU cores

Goal. The adoption of threads requires particular attention in addressing how frequently they have to be executed, so that resource sharing is fair and efficient among threads in each core. With a *polling* strategy, the acquisition thread fetches data from the RSS queues as soon as they are presented by the NIC. This improves timestamping accuracy, but never let the thread sleep with potentially wasting CPU cycles in a busy-loop when no packet is present. A complementary strategy is to enforce *periodic* execution, which allows the system to effectively *share* CPU resources between acquisition and processing threads (Fig. 3d). Yet, this may cause *packet reordering* if the packets of the same flow sit in different RSS queues for too long, or, worse, *losses* in case of suboptimal tuning.

Proposal. We suggest the use of the `SCHED_DEADLINE` (SD) non-default operating system scheduling strategy offered by the Linux kernel. SD guarantees the scheduling of a thread

[4]http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/

within a configurable deadline δ, resulting in a quasi-periodic execution.[5] With appropriate sizing, a single CPU core can be *shared* among two threads, with packets timestamping accuracy and reordering that are under control. To the best of our knowledge, we are the first to investigate the application of SD for packet processing.

D. Flow management and garbage collection

Goal. Stateful per-flow analysis requires garbage collection. In fact, flows may terminate without observing explicit "signaling" packets. This means that a timeout policy needs to be enforced: if no packets are observed for a certain amount of time T_{out}, the flow is considered terminated. It follows that every ΔT (order of seconds) the all F flows (order of millions) in the flow table are checked to verify if they need to be purged. To avoid blocking the packet processing, a natural solution would be to implement the garbage collection in a separate thread. However, this is impractical due to the massive requirement of synchronization primitives it would entail, beside further complicating threads scheduling.

Proposal. We propose to divide the monolithic garbage collection operation in smaller parts. Assuming there are F flows to check every ΔT, we split the operation in M steps, each checking F/M flows, and invoking the garbage collection loop every $\Delta T/M$ time intervals. We report a sensitivity analysis in Sec. V-B.

IV. Experimental setup

Engineering and calibrating a software testbed capable to achieve 40 Gbit/s is not trivial. In our case, a Traffic Generator (TG) allows to replay pcap traces at difference speed, and it is directly wired to a System Under Test (SUT) where we run DPDKStat. For every run, the *sustainable rate* R is empirically measured by looking for the maximum sending rate TG can generate (progressively increasing the sending rate at 100 Mbit/s unit) without observing any packet drop at SUT. We declare that SUT achieves a rate R if in 5 separate runs at the same speed we do not observe losses.

We consider two different SUT: sut-SMP ($\approx$1,500 USD) is a single CPU architecture equipped with an Intel Xeon E3-1270 v3 @3.5GHz, with 4 physical and 4 virtual cores, launched in 2013. It hosts 32GB of DDR3-1333 RAM; sut-NUMA ($\approx$3,500 USD) is a NUMA architecture equipped with 2 Intel Xeon E5-2660 @2.2GHz, each with 8 physical and 8 virtual cores, launched in 2012. Each CPU is equipped with 64GB DDR3-1333 RAM. Both SUT are equipped with 4 Intel 82599 10 Gbit/s Ethernet NICs, connected via a PCIe-3.0 with 16 lanes (64 Gbit/s raw speed).

TG has the same hardware configuration of sut-SMP but has also 8 SSD disks in RAID-0 to speed up pcap files processing. To replay the traffic and control the sending rate,

[5] `SCHED_DEADLINE` guarantees also the periodic thread to not consume more than a fraction of the period via the parameter `sched_runtime`. In our system, we limit CPU time to be shorter than 10% of the period, resulting in a stable system.

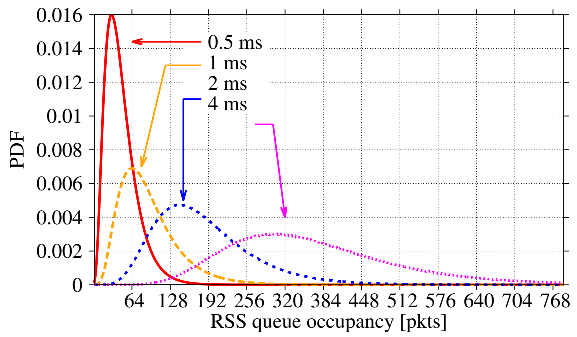


Fig. 4: Distribution of the RSS queue occupancy for varying `SCHED_DEADLINE` packet acquisition intervals δ (sut-NUMA with ISP-80).

we develop our own solution based on DPDK.[6] We aim to benchmark our system using a workload similar to real scenarios. For this reason we rely on replaying packet traces rather than using synthetic traffic generators. However, public available traces usually do not carry payload for privacy issues, hence they are not the optimal choice to test a STA. Conversely, we consider different traces from two operational networks: Campus is a 2 h trace collected in 2015 from Politecnico di Torino campus network ($\approx 10,000$ users, 7.6 M TCP and 5.4 M UDP flows, with average packet size of 811 bytes); ISP-full is 1 h trace collected in 2014 from a European ISP PoP ($\approx 20,000$ residential ADSL users, 3.1 M TCP and 7.7 M UDP flows, with average packet size of 716 Bytes). All traces have been collected during peak time. More details are also available in [6].

Notice the different mix of TCP and UDP between the two scenarios, which results in two complementary benchmarks for the STA. In fact, UDP traffic does not require a very complex state machine, but Bittorrent traffic (popular in ISP-full) results in a huge number of flows.

V. Hardware and Software tuning

We now present experimental evidences of the design principles previously illustrated. We focus on two representative aspects concerning hardware and software that are of general interest, namely tuning packet acquisition, and idle-flow management.

A. Packet acquisition

RSS queues are an instrument that needs to be carefully dimensioned. On the one hand, large RSS queues are needed to avoid overflow and packets loss. Thus we set the RSS queues to the maximum size (4096 packets). On the other hand, since packets are extracted from the RSS queues in batches, we need to control timestamp errors and avoid packet re-ordering.

We argued that is advisable to use a `SCHED_DEADLINE` (SD) kernel policy, which unfortunately induces a non-trivial sampling of the RSS queue size, as the scheduling is not

[6] https://github.com/marty90/DPDK-Replay

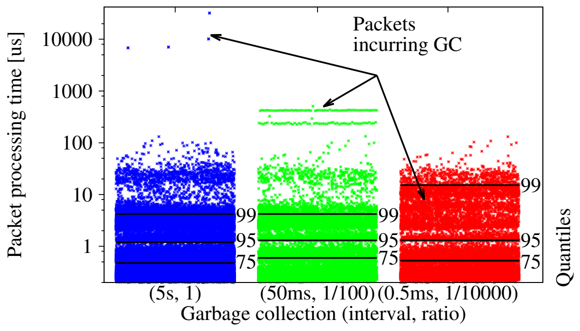


Fig. 5: Per-packet processing time for various settings of the garbage collection period and size (sut-NUMA with ISP-80).

strictly periodic. Fig. 4 reports the empirical Probability Density Function (PDF) of the RSS queue size sampled when the packet acquisition thread is woken up by the kernel: we collect 10 million samples for deadline values of $\delta \in \{0.5, 1, 2, 4\}$ ms when processing 10 Gbit/s traffic. By design, $\delta = 0.5$ ms interval should guarantee sub-millisecond timestamp precision, which is accurate for most cases.

Now, consider packet reordering. Let us suppose client requests and server responses are received at NIC-i and NIC-j, respectively. The per-flow RSS mechanism exposes them consistently to the same process. However, if the packet acquisition thread visits first NIC-j and then NIC-i, an artificial out-of-sequence would be generated. To avoid this, one must guarantee that the processing period of RSS queues is shorter than the client-server RTT, so that client packets are already being removed from NIC-i when server packets are received at NIC-j. With practical Internet RTT that are higher than 1 ms, a deadline of 0.5 ms makes this event very unlikely.

Finally, tail of RSS occupancy distribution is especially important as it correlates with packet losses. With RSS queues of 4096 packets (the maximum allowed), we never recorded any loss in our (relatively short) tests. Yet we can estimate the loss probability. Rather than modeling the packet arrival process at the RSS queue, we opt for a macroscopic approach, and fit the RSS queue size observations in Fig. 4 with an analytic model. We found a lognormal distribution having a good agreement with the experimental data. From the lognormal fit, we can extrapolate the RSS queue overflow probability, i.e., P(Q>4096). For $\delta = 4$ ms, this happens with probability $7.2 \cdot 10^{-10}$. By reducing δ to 0.5ms, the overflow probability becomes smaller than 10^{-20}.

B. *Bounding packet processing time*

Large packet processing time has a particularly severe effect since, during such time, packet loss can happen in the large buffer. In Fig. 5, we report packet processing time samples, when no particular optimization is introduced (blue points - left part of the figure): clear and periodic outliers appear with packet processing time up to 10 ms. These are due to garbage collection (GC) operation that happens periodically.

To control the occurrence of outliers, we divide the monolithic GC in smaller fractions that occur more often. Denoting with $(\Delta T, M/F)$ the GC settings, Fig. 5 shows the original setting (5 s, 1) that scans the entire flow table every 5 seconds, and two settings where both the period and the fraction are divided by the same factor: namely, $100\times$ in the (50 ms, 1/100) case and 10,000 in the (0.5 ms, 1/10,000) case. The plot reports horizontal reference lines for 75th, 95th and 99th percentile statistics computed over 10^6 samples.

Comparing (5 s, 1) to (50 ms, 1/100) we see that the outliers become more numerous (by a factor of 100) but the maximum processing time reduces (roughly by the same amount). Outliers disappear for (0.5 ms, 1/10,000), which happens since the number of flows to be processed by each GC event is now small enough. Observe that the 99th percentile grows, which happens since the number of GC events is large enough to impact the 99th percentile. In a nutshell, the per-packet maximum processing time is now bounded, and exploiting a large buffer between acquisition and processing (Fig. 3c-d) allows to absorb processing jitters.

VI. Experimental Results

We now experimentally evaluate the final DPDKStat design on different systems and configurations.

A. *Periodic acquisition and hyper-threading*

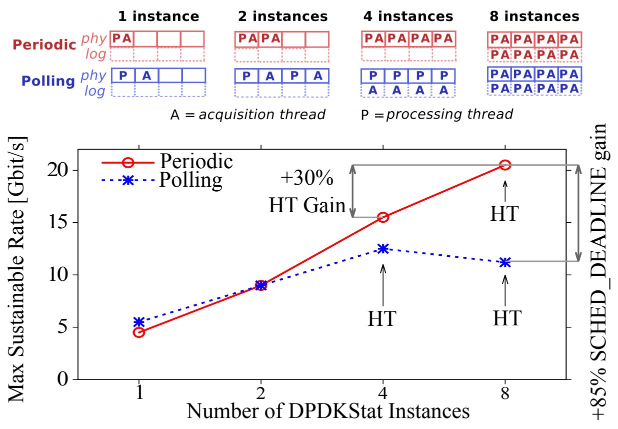
Let us focus on sut-SMP first. Fig. 6a shows the maximum sustainable rate versus the number of parallel processes. Results compare *polling* (dashed line) with the SD *periodic* (solid line) packet acquisition policies. Policies have a direct impact on the how to bound processes to the available cores. In particular, as sketched on the top part of Fig. 6a, when using polling, the best performance is obtained when packets acquisition (A) and processing (P) threads run on dedicated cores (either physical or logic), while it is counter productive if the two threads share the same core. This instead does not occur when using the SD policy.

Both policies present similar performance up to 2 instances, with a small advantage for polling in the single instance case (as 2 physical cores are used). When using more instances, SD presents large performance improvement with respect to polling, a trend maintained also at full capacity. Overall, the system achieves 21 Gbit/s throughput without losses, about twice as much as system performance under polling. This is important to highlight since the system only has only 4 physical cores.

Hyper-threading (HT) yields also remarkable performance speed-up. Compare the 4 vs 8 instances under periodic SD acquisition: running twice as many instances in the same amount of silicon yields +30% performance improvement. Conversely, HT gains are limited using polling. Despite hyper-threading yields benefits in the 4 instances scenario, gains are completely offset in the 8 instance scenario due to increased contention. This confirms that polling is not the best strategy for packet acquisition if the SD policy is available.

B. *Combining different CPUs*

We now consider sut-NUMA where NICs are connected to CPU1. In our setup, all four 10 Gbit/s interfaces are connected to the same I/O Hub and then to the same CPU.



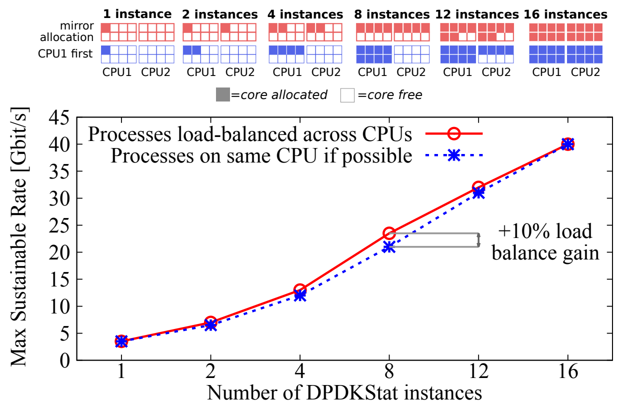


Fig. 6: DPDKStat processing rates using ISP-full trace: (a) sut-SMP and (b) sut-NUMA (no hyper-threading)

In this scenario, we have an additional degree of freedom in terms of core allocation policies. As schematically represented in top of Fig. 6b, we can either (i) use all cores of CPU1 (dashed line), which is closer to the NICs, or (ii) balance the load across CPUs (solid line). In these tests, hyper-threading is disabled and we run all processes on the 16 physical cores.

As for the previous analysis, throughput scales linearly with the numbers of cores, and the system successfully reaches 40 Gbit/s with no packet losses. Interestingly, the system is slightly faster when allocating processes on both CPUs rather than filling CPU1 first (up to +12% in the 4 instance scenario). Potentially the system could be able to process even more traffic (e.g., enabling HT) but unfortunately we cannot test this hypothesis since (i) our testbed is limited to 40 Gbit/s and (ii) Intel NICs offer a maximum of 16 RRS queues (thus maximum of 16 processes). We can however assess HT gains to hold: in particular, when binding all 16 processes to run only on the 8+8 cores of CPU1 with HT enabled, we achieve 24 Gbit/s, corresponding to a +20% of performance improvement with respect the 8 instances scenario reported in Fig. 6b. This gain is lower than what obtained from sut-SMP, possibly due to the different HW specs. Even if not possible with our hardware, it would be interesting to check different allocation policies where multiple NICs are connected to different I/O Hubs and CPUs.

VII. Conclusions

We reported our experience in the design, implementation and benchmarking of a system for traffic analysis to process 40 Gbit/s with COTS hardware. We argued that applications must leverage large intermediate buffers to absorb variable processing times and avoid packet losses. We found periodic packet acquisition policies to be preferable over traditional polling solution, with the SD scheduler offered by the Linux kernel being amenable of precise buffer control with no packet losses when properly combined with RSS queues. Hyper-threading offered a sizable gain (20%-30%), while process allocation over multiple NUMA nodes furthered improve performance (10%). Overall, we demonstrated that with a careful design is possible to achieve multi 10 Gbit/s without specialized and expensive hardware.

Acknowledgments

This work was carried out at LINCS (http://www.lincs.fr/) and was supported by the WWTF Agency through the BigDAMA project and by NewNet@Paris, Cisco's Chair "Networks for the Future" at Telecom ParisTech (http://newnet.telecom-paristech.fr/).

References

[1] X. Chen *et al.*, "Para-Snort: A Multi-thread Snort on Multi-core IA Platform," in *PDCS*, 2009.
[2] M. A. Jamshed *et al.*, "Kargus: A Highly-scalable Software-based Intrusion Detection System," in *ACM CCS*, 2012.
[3] K. NamUk *et al.*, "A Scalable Carrier-Grade DPI System Architecture Using Synchronization of Flow Information," *IEEE JSAC*, vol. 32, no. 10, pp. 1834–1848, Oct 2014.
[4] L. Koromilas *et al.*, "Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures," in *ACM/IEEE ANCS*, 2014.
[5] A. Finamore *et al.*, "Experiences of Internet Traffic Monitoring with Tstat," *Network, IEEE*, vol. 25, pp. 8–14, 2011.
[6] M. Trevisan *et al.*, "Dpdkstat: 40gbps statistical traffic analysis with off-the-shelf hardware," in *Tech. Rep.*, 2016. [Online]. Available: https://www.telecom-paristech.fr/~drossi/paper/DPDKStat-techrep.pdf
[7] T. Barbette *et al.*, "Fast Userspace Packet Processing," in *ACM/IEEE ANCS*, 2015.
[8] V. Moreno *et al.*, "Testing the capacity of off-the-shelf systems to store 10gbe traffic," *IEEE Communications Magazine*, vol. 53, no. 9, pp. 118–125, 2015.
[9] H. Jiang *et al.*, "Scalable High-performance Parallel Design for Network Intrusion Detection Systems on Many-core Processors," in *ACM/IEEE ANCS*, 2013.
[10] A. Das *et al.*, "An fpga-based network intrusion detection architecture," *IEEE Transactions on Information Forensics and Security*, vol. 3, no. 1, pp. 118–132, 2008.
[11] K. Jaic *et al.*, "A practical network intrusion detection system for inline fpgas on 10gbe network adapters," in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 2014, pp. 180–181.
[12] G. Bianchi *et al.*, "Streamon: A software-defined monitoring platform," in *ITC*, 2014.
[13] L. Deri *et al.*, "ndpi: Open-source high-speed deep packet inspection," in *TRAC*, 2014.
[14] W. Shinae and P. KyoungSoo, "Scalable TCP session monitoring with Symmetric Receive-Side Scaling," *KAIST, Daejeon, Korea, Tech. Rep*, 2012.
[15] J. Lelli *et al.*, "Deadline scheduling in the linux kernel," *Software: Practice and Experience*, 2015.