DPDKStat: 40Gbps Statistical Traffic Analysis with Off-the-Shelf Hardware

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

March 11, 2016

Abstract

In recent years, advances in both hardware and software offer to user-space applications O(10Gbps) worth of traffic. Processing data at such line rate with software running on Commercial Off-The-Shelf (COTS) hardware requires careful design. In the literature, this challenge has been extensively studied for Network Intrusion Detection Systems (NIDS), with several proposals achieving scalability by exploiting GPU and NPU multi-core architectures. Yet, NIDS are relatively "easy" to parallelize given their core functionality is to match the input traffic with respect to a large catalog of rules.

In this work, we instead focus on more generic Statistical Traffic Analysis (STA) tools. Specifically, we target the design of a passive monitoring solution that extracts hundreds of per-flow metrics collected in different output formats (text logs, RRD database, empirical distributions, etc.). Differently from NIDS, such metrics normally provide an extensive vision of specific traffic dynamics (e.g., TCP anomalies, video streaming QoE, HTTP content delivery, etc.), resulting in a few analysis engines, but an intrinsically more pipelined workflow. Parallelisation is then less trivial than for NIDS.

We propose DPDKStat, a high speed STA that combines the Intel DPDK framework with Tstat, a passive traffic analyzer, to achieve 40Gbps of line rate processing. Beside reporting a detailed evaluation of the system using real traffic traces, we contribute an in-depth discussion on the design challenges to achieve scalability, and the lesson learned we believe of common interests for the traffic monitoring research community.

1 Introduction

The last years have witnessed a growing interest towards multi-core architectures for Internet traffic processing. Indeed, on the one hand the Moore law scales at a lower pace [3] compared to the +50% annual bandwidth consumption rate found by recent estimates [1]. On the other hand, nowadays Internet traffic is overcrowded of services requiring more sophisticated and powerful tools for their monitoring. In addition to the intrinsic difficulty of parallel programming [26], multi-core Internet traffic processing is further complicated by the need to acquire, move, and process *packets*, while maintaining their logical organization in *flows*. These are daunting tasks to tackle at O(10Gbps) line rate.

The advent of open source (e.g., Intel DPDK, PacketShader, netmap, netapi, PFQ, etc.) and proprietary (e.g, libDNA) advanced software packet acquisition libraries, not to mention ad-hoc hardware solutions (e.g., Endace DAG, NapaTech, etc.) alleviates the problem of the mere data acquisition: these solutions enable O(10Gbps) packet access rates in user-space with zero-copy (i.e., packets are moved via DMA from the NIC to the kernel, enabling visibility of such memory also to userspace to avoid extra data copy).

To process such deluge of data, software developers have embraced multi-core CPUs, and (possibly) massively parallel Graphical Processing Units (GPU) or Network Processing Units (NPU)

architectures. This is testified by seminal [14, 6] and more recent works [34, 16, 18, 25, 20, 24] successfully scaling and optimizing multi-core Network Intrusion Detection Systems (NIDS). Notice however that NIDS tools are intrinsically parallelizable, given that their function is to test a set of rules on individual packets. Conversely, Statistical Traffic Analysis (STA) tools offers an in-depth vision on specific traffic classes (e.g., performance for video streaming, HTTP content delivery, or more basic TCP statistics like RTT, congestion window, etc.) using few (and pipelined) analysis engines. Hence, are less interesting (and trivial) to parallelize.

In this work, we propose DPDKStat, a system combining the Intel DPDK [12] framework for packet acquisition, and Tstat [8] a STA that builds, keeps, and updates hundreds of per-flow statistics. Overall, our work show that it is possible to achieve 40Gbps of traffic processing on a Commercial Off-The-Shelf (COTS) solution that costs less than 4,000 USD. Yet, we highlight that the aim of this paper is not to present "yet another fancy traffic monitoring tool". Conversely, we focus our attention to derive *design principles* and test them using real traffic, sharing the most useful and general *lessons learned* in crafting our solution. Summarizing, our major contributions are:

- We propose a periodic packet acquisition policy (leveraging the recent `SCHED_DEADLINE` Linux scheduling discipline, never considered in previous works), performing a in-depth analysis to limit timestamp error and avoid packet reordering and losses.
- We study the system vertical scalability, using real traffic as a benchmark: notably, we quantify benefits of periodic packet acquisition (gain of 2× over polling), hyper-threading (+20-30%) and load balancing over multiple CPU nodes (+10%).
- We offer both our DPDKStat as well as our traffic generator, able to respectively analyze and reply traffic at 40Gbps, as open source software at [33, 2].

In the following, we first provide a broad picture of the related work in the area (Sec. 2) to better motivate and contrast our work with the current state of the art. We next present the design principles behind DPDKStat (Sec. 3), followed by details of our benchmark procedure and testbed setup (Sec.4). We instantiate these principles by calibrating hardware and software (Sec. 5), on which we perform an extensive experimental evaluation (Sec. 6). Finally, a summary of lessons learned is discussed along with open points (Sec. 7).

2 10 Years of High Speed Traffic Processing Solutions

The last decade witnessed a flourishing interest for high speed Internet traffic processing. Since seminal works [14, 6] coping with few hundreds of Mbps line rate, different solutions breaking the 10 Gbps "barrier" have been proposed by both academia [34, 16, 18, 25, 20, 22, 11, 24, 5] and vendors [28, 27, 13]. The first challenge, namely how to efficiently transfer packets from the Network Interface Cards (NIC) to the main memory, has been solved by advanced packet capture libraries (compared and benchmarked in [5, 10]). Processing scalability instead can be achieved adopting multi-cores technologies such as Non-Uniform Memory Access (NUMA), GPU and NPU, and Field-Programmable Gate Array (FPGA). Recent proposals further push systems design envisioning Hadoop-based [19] or stream-based [31] solutions to process raw packets.

All these efforts result in a very tangled and overcrowded landscape of options. To testify such complexity, Tab. 1 summarizes the characteristics of the most prominent works on the subject spanning over the last decade. Despite a few works provide in depth analysis of specific aspects (e.g., benchmarking advanced packet capture libraries [5, 10], load balancing [34, 25, 17, 21], and energy consumption [21]), given the variety of the adopted hardware, software, and input traffic, any comparison can merely have *qualitative* value.

For the sake of illustration, we represent in Fig.1 proposals as circles, centered at $(rate, year)$, with a radius proportional to the number of cores used to achieve the advertised processing rate.[1] A

[1] Despite proportion are respected, we assign a smaller weight to GPU cores since are less powerful than CPU core.

	Proposal			Rate [Gbps]		Software				Hardware		
	Yr	Name	Ref	Real	Synth	Software	#Rules	Load bal.	Pkt lib	CPU	GPU	Specifications
NIDS	06	Intel	[14]	0.1	0.5	Snort	n/a	SW 4-tuple	n/a	4		2x 2-Core Xeon LV
	09	ParaSnort	[6]	0.8	0.5	Snort	10,000	JSQ⋆	n/a	8		2x 4-core Xeon E5335
	11	MIDeA	[34]	5.2	7.6	Snort	8,192	RSS+dyn CPU	PF_RING	8	480	2x 4-core Xeon E5520 [2x NVIDIA GTX480]
	12	Kargus	[16]	25.2	19-33	Snort⋆	3,111	RSS+dyn GPU	PSIO	12	1024	2x 6-Core Intel X5680 [2x NVIDIA GTX580]
	13	*unnamed-A*	[18]	-	7.2-13.5	Suricata	7,571	JSQ	Tilera-mPIPE	36		TileraG 36
	14	DPI-S	[25]	40	-	custom	40,000	custom	Tilera-mPIPE	144		4x Tilera 36
	14	*unnamed-B*	[20]	-	15	OpenCL DPI	10,000	n/a	custom	8	480	1x 4-Core Intel i7-3770 [1x NVIDIA GTX 480]
	15	*unnamed-C*	[24]	-	15-79	Suricata+Kargus	2,435	dyn CPU	Tilera-mPIPE	72+16		Tilera36 + 2x 8-Core Intel E5-2690
others	14	nDPI	[22]	10	-	OpenDPI⋆	170	DNA	PF_RING	2		1x 8-Core Intel i7
	14	StreaMon	[11]	1.9-6.47	-	custom	-	SW 4-tuple	PFQ	6		1x 6-Core Intel Xeon X5650
	15	FastClick	[5]	-	40	Click⋆	-	RSS	netmap,DPDK	12		1x 6-Core Intel Core i7-4930K (Hyper-threading)
	15	DPDKStat	-	40	-	Tstat	Fig. 4	RSS	DPDK	16		2x 8-Core Intel Xeon E5-2660
vendor	13	HyperScan	[13]	160	-	custom	n/a	n/a	n/a	16		2x 4-Core Intel Xeon E5-2600 (Hyper-threading)
	14	Procera	[27]	-	≈40	custom	n.a.	n/a	n/a	36		2x 18-Core Intel XeonE5 (Hyper-threading)
	15	Sandvine	[28]	>40	-	custom	n/a	n/a	n/a	>40		cluster of Tilera-like blades

⋆ = modified version

Table 1: State of the art parallel NIDS and traffic analysis processing (this work highlighed)

straight line (in semilog scale) represent a Moore's-like exponential increase of raw processing rate, doubling every year from the initial starting point of O(100Mbps) processing rate [14]. Comparison with "historical" work such as [14, 6] is only anecdotal: specifically, 2015 processing rate exhibit a speedup close to 2^{10} (2^6) with respect to the 2006 [14] (2009 [6]), quite well matching Moore expectations[2].

Notice that most of the works in Tab. 1 investigate NIDS tools performance. Those tools are designed to trigger alarms when the input traffic matches signatures from a predefined dictionary, i.e., they report concise information about the input traffic activity. Given the limited need to handle per-flow state statistics (analysis are operated atomically on per-packet base or using compact state machines) they are "easily" amenable to parallel operation (Suricata is multi-threaded since the first release, while Snort will be from v3, in alpha release as we write). Pattern matching is however costly (e.g., Suricata and Snort can cope with only O(100Mbps) per-core [29, 15]): hence, NIDS scalability is achieved with a large number of GPU [34, 16, 20] or NPU [20, 25] cores (i.e., the large circles in Fig.1).

To the best of our knowledge, less effort has been devoted in studying scalability issues for STA. Tab. 1 also include high-speed tools that, despite not being STA, are not pure NIDS either: namely, StreaMon [11] is a SDN traffic monitoring framework, FastClick [5] is an advanced software router based on Click, while nDPI [22] is a pure traffic classifier derived from OpenDPI. Other works discuss scalability addressing very simple operations like counting packets [9]. Yet, these tools are clearly not fully-fledged STA.

We provide a more in-depth analysis of both NIDS and STA specificities, as well as its implication on system design, in Sec. 3.1. We anticipate that STA comprises a smaller, yet more varied, set of function intrinsically more difficult to parallelize compared to NIDS: specialized STA functions indeed share per-flow *state*, leading to a more pipelined analysis workflow than for NIDS. Yet, as we see from Tab. 1, the current state of the art is scattered along many directions, so that is difficult to even qualitatively compare two designs, let alone to learn useful design guidelines that are not tool-specific, or that go beyond the NIDS perimeter.

These issues raise questions concerning parallel STA software design, which we address in this work, such as: How to avoid locking? Are NUMA architectures sufficient? Is there any advantage is using hyper-threading? How to allocate processes/threads on each core?

3 System Design

To design and implement any parallel network traffic processing tool, the type of analysis to perform cannot be left out of consideration. In this section we try to address this challenge by comparing the footprint of Tstat, our reference STA implementation, with respect to Bro, Snort and Suricata (Sec.3.1). We next present the design questions we explore, and principles we follow, in engineering DPDKStat, warning about pitfalls and illustrating guidelines to achieve lossless monitoring at 40Gbps (Sec.3.2).

[2] We intent here Moore in a broad sense [3]

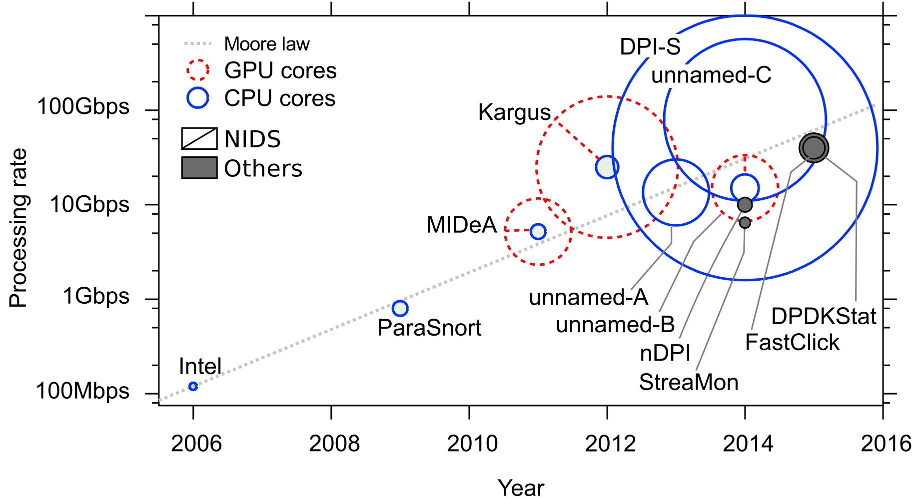

Figure 1: Synoptic of related work surveyed in Tab.1. Circles are centered on the year and processing rate. Radius size is a logarithmic scaling of the number of cores employed by the system.

3.1 NIDS and STA Characteristics

Qualitative comparison. STA and NIDS tools address different traffic monitoring functions. STAs have the goal of processing, extracting and collecting both *per-packet* (e.g., anonymization of IP addresses via cryptographic functions, counting number of IP/TCP/UDP packets observed, packet length distribution, etc.) information, as well as *per-flow* statistics (e.g., three-way-handshake time, amount of data carried in a TCP flow, TCP events such as TimeOut and FastRetransmit, min/ max/avg RTT, TLS negotiation delay, video streaming QoE and content properties, etc.). This entails a stateful monitoring, where each TCP, UDP, and possibly layer-7 protocol states need to be tracked. Overall, the number of per-flow statistics can easily be very large (>100) thus generating quite a large amount of output that need to be saved on disks. Efficient memory management is a must too, with garbage collection that must be periodically executed to purge those flows that have become inactive.

Conversely, NIDS tools trigger (typically few) alarms when the observed traffic matches some rules. Alarms needs to be logged, producing per-event rather than per-flow entries. Both per-packet and per-flow rules are considered, thus entailing per-flow management as for STA. However, Deep Packet Inspection (DPI) is the core feature of NIDS and not statistics. As such, NIDS design is tightly optimized to handle a large amount of rules, i.e., $O(10^4)$. Since rules can be parallelized, performance greatly benefits from GPU/NPU support (see Sec. 2).

DPI is commonly used in STA for traffic classification too: however, the number of rules is way smaller than for NIDS, i.e., $O(10^2)$. Thus, making GPU/NPU support less appealing. Conversely, STA tools provide deeper insights for several traffic classes, by extracting a large number of metrics at multiple layers of the protocol stack: for this reason STA tools normally have fewer analysis engines, arranged in a more pipelined workflow.

Some NIDS tools can also operate as an Intrusion Prevention System (IPS), i.e., they forward only legitimate traffic. In such cases, packet forwarding capability is required, whose cost can be (again) easily parallelized [5]. STA tools instead do not require such a functionality.

To achieve high performance, the preferred programming languages are C (Tstat, Suricata, Snort) or C++ (Bro), with possibly some CPU intensive parts optimized in assembly. Dedicated high-level languages instead incur severe performance impairment [32].

Quantitative comparison. To better quantify the different processing requirements, Fig. 2

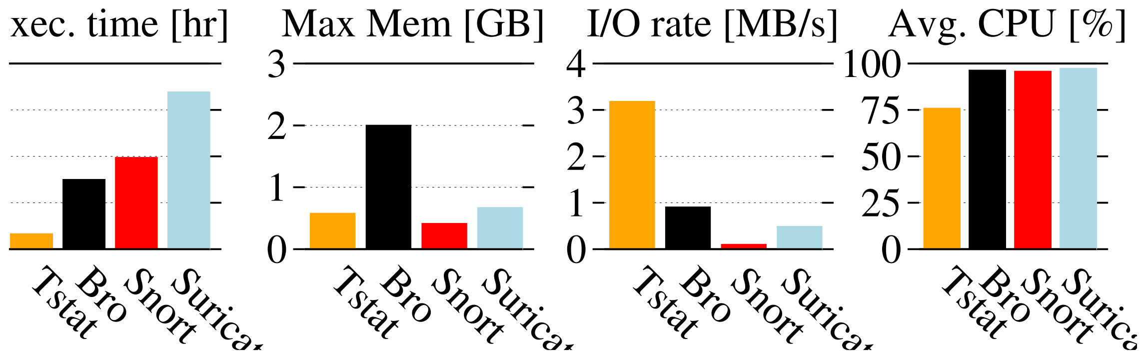

Figure 2: STA and NIDS performance comparison (1-core, all tools with default configuration).

compares processing time, maximum memory, I/O rate, and average CPU utilization when running Tstat, Bro, Snort and Suricata on the same benchmarks. Our intent is not to provide a punctual comparison among the tools, but to give the reader a *high-level quantification* of performance requirements. We run all tools with default settings, with Snort and Suricata sharing the same rules set (a catalog of 73,012 Emerging-Threat rules). Tests run on the same hardware using a single CPU core to process about 1 TB of traffic read from the local disk (details about the sut-SMP hardware and Campus trace are reported in Sec. 4).

Tstat is the fastest tool, processing the entire trace in $\approx$ 20min ($\approx$ 6 Gbps), 4.5x faster than Bro (the best performing NIDS) and 10x faster than Suricata (the slowest in the benchmark[3]). Maximum memory usage is comparable, and limited to less than 2GB. Considering the amount of output produced, the I/O rate plot shows that Tstat generate 3.5$\times$ more data per unit of time than NIDSs. Indeed, the output formatting, and the per-flow I/O operations are responsible for the limited average CPU utilization, which is capped at 75% for Tstat, as show in the rightmost plot.

In summary, STA tools are fairly lighter than NIDS from a mere computation intensive point of view, with a short per-packet processing pipeline, that questions the adoption of GPU/NPU. They require to compute a heterogeneous set of metrics, that makes multi-core SMP and NUMA architectures interesting alternatives.

3.2 Design Principles and Lesson Learned

Several considerations hold in the design of a multi-core enabled tool. Some choices are tool dependent (e.g., managing output), while other principles are, we believe, novel and general (e.g., controlling packet acquisition via OS periodic scheduling). In this section we provide an overall discussion of design guidelines, that we phrase as a Q & A in the narrative, outlining goals and the proposal to achieve them. We instead defer details, parameter tuning and performance to Sec. 6.

We assume the STA application runs on a COTS hardware, and has to monitor traffic flowing on 10Gbps link(s), whose traffic is mirrored using inexpensive optical/electric taps toward the processing system NICs. Notice that two NICs are required for each single full-duplex link. To cope with the load, traffic is then split among different processing engines that are bound to different CPU cores. This calls for resource allocation among different processes or threads. In Fig. 3, we assume to have n NICs, and c CPU cores at disposal.

Goal: Packet acquisition and per-flow load-balancing. Several solutions have been recently proposed to provide efficient packet acquisition on COTS hardware, which all solve the problem of efficiently moving packets from the NICs, and that are contrasted in [5, 10]. However, to

[3]Suricata execution time is only halved when running with 16 threads, 2 for each core.

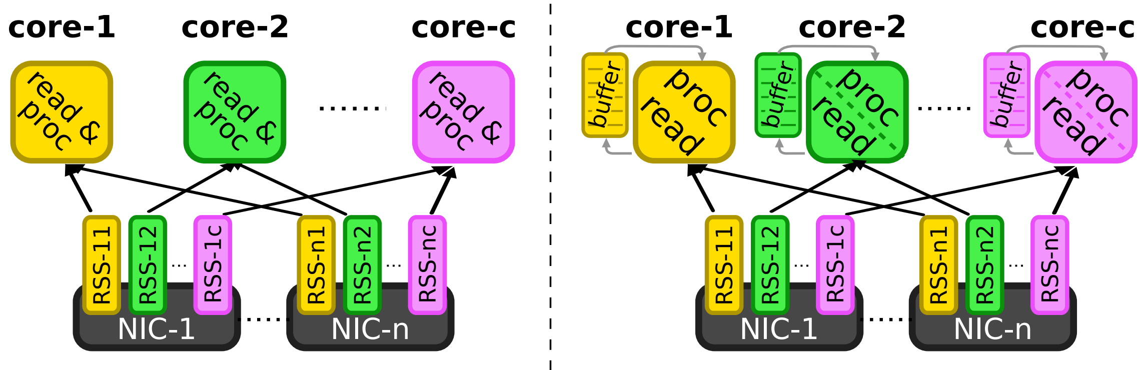

Figure 3: System architecture. Direct RSS access (left) and buffered access (right).

compute per-flow statistics, we need to correlate packets received irrespective of the NIC where the packets are observed: hence, the packet acquisition library need thus to offer a *flow-preserving load balancing function* for correct traffic processing. This offers also the appealing opportunity to split the traffic to be processed among c CPUs by using multi-process approach, avoiding costly synchronization primitives [30].
Proposal: Hardware supported load-balancing. We investigated libDNA (now evolved into the PF_RING ZC library[4]) that offers the possibility to run custom packet *load balancing in software*, via the so called "DNA cluster". In this case, all packets received from any NIC are passed to the DNA cluster process, which then (i) timestamps and (ii) forwards them to the correct processing engine. Unfortunately a libDNA prototype, made by a simple load-balancer followed by processing engines that just count packets, limitedly achieve 6 Gbps processing rate in our tests: the software load-balancing process constitutes the main system bottleneck (details in [4]). However, modern NICs offer (limited) *hardware load balancing*, e.g., via the Intel Receiver Side Scaling (RSS) queues supported by the DPDK library. Consistent per-flow load balancing is possible with specific hashing function [35]. This results in a system where packets arriving at NICs are hashed to RSS queues, from which they are then extracted and processed by the STA. This is shown in Fig. 3, where each analysis application extracts packets for the RSS queues (in this scenario, the number of RSS queues is equal to the number of CPU cores). RSS queues are however a scarce resource and we provide an in-depth sensitivity analysis of their tuning in Sec. 5.1.

Goal: absorbing traffic and processing jitters. The per-packet analysis time of STA (and NIDS) is not constant. If on the one hand traffic processing tools need to be engineered to minimize the average packet processing time, unexpected (large) processing delays (due to I/O, periodic operations, critical packet composition, etc.) can lead to losses. Similarly, unexpected traffic bursts can lead to losses too.

Packet acquisition libraries implements circular buffers to absorb such jitters. Yet, 1MB of buffers only absorb *less than one millisecond* worth of traffic at 10Gbps causing thus possible packet drops. This is a new problem that emerges specifically at very high speed, and was indeed previously ignored.
Proposal: use a lock-free large packet buffer. Our solution is to introduce to each analysis module a *large buffer*, as reported in the right-hand side of Fig. 3. We consider a buffer of 1 GB for each analysis process, i.e., sufficient to store *almost one second* of 10 Gbps traffic. This calls for a system in which packet acquisition and analysis threads are decoupled: (i) an "acquisition" thread extracts packets from the RSS queue, timestamps and enqueues them to the tail of the buffer; (ii) a "processing" thread dequeues packets from the buffer head, and analyses them. Normally

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

such design choice would lead to expensive resource access synchronization. Furtunately, the Intel DPDK framework offers lock-free shared buffer data structures using state of the art zero-copy data acquisition[5] that fit our requirements. However, this complicates the CPU sharing design, since now at least two separate threads are needed for each processing applications.

Goal: efficient sharing of CPU cores. The design results now into a *hybrid* approach: (i) different independent processes are attached to (a group of) RSS queues, but (ii) each process has separate threads managing the packet acquisition and analysis functions. This calls for particular attention in addressing *how frequently* the two threads have to be executed. One option is to fetch data from the RSS queues as soon as they are presented by the NIC, via a *polling* packet acquisition thread. This improves performance and timestamping accuracy[6], but never let the thread idle, wasting CPU cycles in a busy-loop. A complementary strategy is instead to enforce *periodic* execution, which allows to effectively share CPU resources between both threads. Yet, this may cause packet reordering (due to packets sitting in the RSS queues for long time before being fetched) or, worse, losses in case of careless tuning.
Proposal: use non-standard OS scheduling disciplines. We investigate and compare both packet acquisition strategies: For *polling*, a CPU core is allocated to the packet acquisition thread, which keeps looping on its set of RSS queues, extracts all present packets, timestamps and moves them in the large buffer; the processing thread must then run on a different CPU core. *Periodic scheduling* relies instead on a novel process scheduling discipline featured in Linux kernel from version 3.14, namely `SCHED_DEADLINE` (SD). Interestingly, SD guarantees the scheduling of a thread within a configurable deadline, resulting in a quasi-periodic execution compared to the default Completely Fair Scheduler (CFS) policy, which approximate fair time sharing among threads. With appropriate dimensioning, the same CPU core can be shared among the two threads, with packets timestamping accuracy and reordering that are under control, increasing efficiency with respect to polling at the same time. To the best of our knowledge, we are the first to investigate SD usage for packet acquisition. We study the benefit of periodic scheduling in Sec. 6.1.

Goal: Flows management and garbage collection. Stateful per-flow management requires flows that are terminated to be correctly managed. Unfortunately, a flow may terminate without observing explicit signaling packets, so that we need to implement a timeout policy: if no packets are observed for a certain amount of time T_{out}, the flow is considered terminated. We need to flush those flows to generate statistics, and to limit memory usage via garbage collection. Such operation is intrinsically periodic: every T, the complete flow data structures with F entries is scanned to check which flows need to be purged, with F in the order of several millions. To avoid blocking the packet processing, a natural solution would be to implement a garbage collection thread. However, this incurs in massive adoption of synchronization primitives that must be invoked for each packet. Moreover, this would further complicate thread scheduling.
Proposal: split garbage collection. We propose a simple, yet effective, strategy that can be operated "in line", i.e., periodically devoting (little) time of the analysis thread for garbage collection. In a nutshell, assuming there are F flows to check every T. We split the operation in M steps, each checking F/M flows, and invoking the garbage collection loop every T/M time intervals: we expect the time devoted to garbage collection to be reduced by a factor M at each call. We investigate garbage collection strategies, presenting a sensitivity analysis of the important aspect needed to properly tune M, in Sec. 5.2.

Goal: managing output. STA applications produce quite a sensible amount of output, which is a possibly slow operation due to I/O access, performed per-flow (e.g, log formatting), or periodically (e.g., consolidating samples in Round Robin Databases (RRD), a de-facto standard solution but also a potential bottleneck[7]).

---

[5] We use DPDK memory pools coupled to a FIFO buffer of pointers that avoid locking via software transactional memory, `http://dpdk.org/doc/api/rte__ring_8h.html`

[6] All packets in the RSS queue are extracted in a single batch. Timestamping is corrected as in [23].

[7] `http://net.doit.wisc.edu/~dwcarder/rrdcache/`

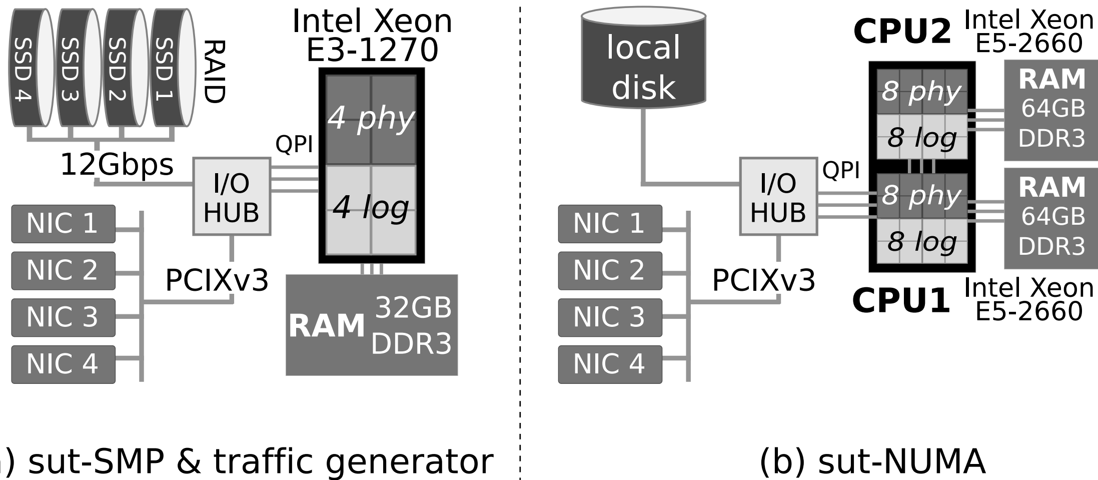

Figure 4: Hardware specs: (left) sut-SMP: 1 CPU, 4 physical+4 virtual cores. (left) TG: same as sut-SMP, with 4 SSD in RAID-0. (right) sut-NUMA: 2 CPUs, 8+8 cores each.

Proposal: keep it simple, stupid. After experimenting with several policies for handling the different output types, we settled to delegate to a separate low-priority thread only the most "bursty" functions that are well suited to be performed as background operations (e.g., such as periodic RRD updates that create or access literally thousands of histogram files). Conversely, per-flow logging is performed at flow finish time, increasing packet processing time of the last packet: the presence of the large buffers allows us to absorb this variability without requiring ad hoc patches.

4 Experimental setup

Testbed configuration. Engineering and calibrating a software testbed capable of 40Gbps benchmarking is not trivial. The testbed comprises a System Under Test (SUT) whose performance are under observation, and a Traffic Generator (TG) able to saturate wire speed. The TG is then connected to the SUT: experiments are run, and the *sustainable rate* is empirically measured by looking for the maximum average sending rate that allows the SUT to process traffic without suffering any packet drop. The TG is able to regulate the average sending rate (by active waiting with busy loop between packets), so that a dichotomic algorithm is used to find the sustainable rate, with precision of 100 Mbps (1%).

System Under Test (SUT). In this work, as SUT we consider two COTS systems. sut-SMP (≈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[8] in 2013. It hosts 32GB of DDR3-1333 RAM. sut-NUMA (≈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[9] in 2012. Each CPU is equipped with 64GB DDR3-1333 RAM. Each system is equipped with 4 Intel 82599 10Gbps Ethernet NICs, connected via a PCIx-3.0 with 16 lanes offering 64Gbps raw speed.

Traffic Generator (TG). For the TG, we use an hardware system equivalent to sut-SMP, equipped with 4 SSD disks in RAID-0 (offering a raw disk read speed of 12Gbps). We develop a novel DPDK-based tool that replays pcap packet traces stored on disk. However, to achieve

[8] `http://ark.intel.com/products/75056`
[9] `http://ark.intel.com/products/64584`

	Flows (M)		Flows per-class (%)				Pktsize
name	TCP	UDP	HTTP	HTTPS	P2P	oth	avg
Campus	7.6	5.4	21.3	22.2	11.5	45	811
ISP-full	3.08	7.76	10.8	8.2	46.2	34.7	716
ISP-80	1.50	-	78.0	0.03	0.2	21.7	909
ISP-N80	1.57	7.76	0.1	9.5	53.6	36.8	610

Table 2: Packet traces.

40Gbps several optimizations are necessary. In particular, the traffic generator sent a modified version of a packet on all interfaces: packets sent to the k-th NIC present source and destination IP addresses increased by k.[10] This generates different flows, that, even if identical, are hashed to different RSS queues, and thus received and processed by the SUT as different flows. We release as open source software at [33].

Packet traces. Tab. 2 details the traces used in this study, that we collected from two different real operational networks: namely, Campus is a 2 hr trace collected in 2015 from Politecnico campus network, with 10,000 users; ISP-full is 1 hr trace collected in 2014 from a European ISP PoP serving about 20,000 residential ADSL customers. We see from Tab. 2 that the traffic mixture is different: we also extract two subsets from ISP-full (namely HTTP vs non-HTTP traffic) to further exacerbate differences in the mixture (e.g., packets and flows size, percentage of UDP traffic, etc.) to gather performance sensitivity to the input workload (Sec.6.3).

5 Hardware and Software tuning

We now instantiate and calibrate the principles early illustrated in our prototype. For lack of space, we focus on two representative aspects concerning hardware and software that are of general interest, namely NIC settings and packet acquisition policies (Sec.5.1), and idle-flow management (Sec.5.2).

5.1 Packet acquisition policies

Sizing RSS queues. RSS queues are a useful instrument, that need to be carefully dimensioned in light of the following tradeoff. On the one hand, large RSS queues are needed to avoid overflow: as packet loss biases measurement results, it must be imperatively avoided. At the same time, as packets are not individually timestamped (as in the case of dedicated capture cards with high-precision clocks): the larger the batch, the larger the imprecision of individual packet timestamps. At last, processing large batches of traffic could artificially generate packet re-ordering when moving packets from different RSS queues into the same large buffer.

We argued that is advisable to use a `SCHED_DEADLINE` (SD) kernel policy to wake up the thread at quasi periodic times, freeing up CPU resources with respect to polling mode (and quantify gains in Sec. 6.1). Yet, SD policy induces a non-trivial sampling of the RSS queue size, as it guarantees that the process will be scheduled once before the deadline, but the scheduling is not strictly periodic. Fig. 5 reports the empirical Probability Density Function (PDF) of the RSS queue size, sampled when the packet acquisition thread is waken up by the kernel: we collect 10 million samples for different deadline values of $\delta \in \{0.5, 1, 2, 4\}$ ms when sending 10Gbps traffic (ISP-80trace) through a single NIC in sut-NUMA. By design, $\delta = 0.5$ ms interval should guarantee sub-millisecond timestamp precision, which is accurate for most cases. The upper x-axis reports the equivalent time packets of 750B size spend in the queue. For $\delta = 0.5$ ms, the bulk of timestamp

[10] We tested other functions to see if this could introduce any bias. No noticeable impact has been observed.

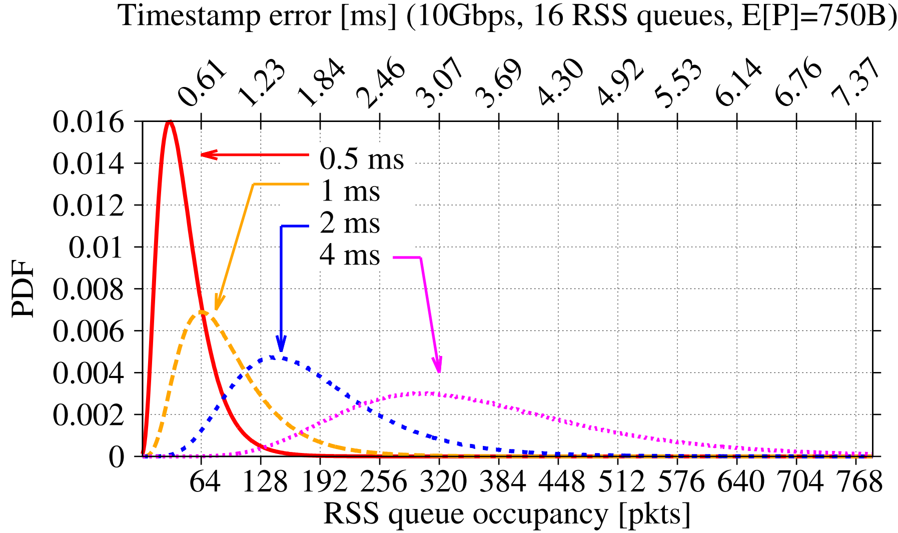

Figure 5: Distribution of the RSS queue occupancy for varying `SCHED_DEADLINE` packet acquisition intervals δ (sut-NUMA with ISP-80).

errors are indeed sub-millisecond, with only about 1% of samples possibly[11] exceeding 128 packets or ≈ 1 ms (see Fig. 6). When large δ values are used, the batch size in the RSS queue grows, thus introducing larger errors.

Considering packet reordering, the scenario which would results very critical is when packets of the *same flow* are processed out-of-sequence. For instance, consider client requests and server responses observed at NIC-i and NIC-j, respectively. The RSS mechanism exposes them consistently to the same process. But if the packet acquisition thread visits first NIC-j and then NIC-i, an artificial out-of-sequence could 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.

Tuning periodic acquisition. RSS occupancy distribution tail is especially important as it correlates with losses of packets at the monitor. With RSS queues of 4096 packets (the maximum allowed), we never recorded any loss in our measurements. Yet we can estimate the loss probability. Rather than modeling the packet arrival process at the RSS queue (complex as it depends on stochastic properties of the traffic at the monitor, the traffic mixture, the RSS hash function, etc.), and modeling the acquisition thread service time (complex as it depends on the scheduler policy which is not strictly periodic, and on external factors such as the presence of other threads active on that CPU, the kind of processing they perform, etc.), we opt for a macroscopic approach. We fit the RSS queue size observation with an analytic model. We find a lognormal distribution having a good agreement with the experimental data: e.g., for $\delta = 4$ ms (the most delicate case as the queue is large and thus where the most precise fit is needed) lognormal parameters $\mu = 5.85$ and $\sigma = 0.40$ exhibit asymptotic errors of 0.008% and 0.094% respectively. Fitting results are reported in Fig. 6.

From the lognormal fit, we can then extrapolate the RSS queue overflow probability, i.e., P(Q>4096). For $\delta = 4$ ms, this can happen with probability $7.2 \cdot 10^{-10}$, i.e., a rare but not impossible event. By reducing δ to 2 ms (or 1 ms) the lognormal model estimate a RSS overflow probability of $7.2 \cdot 10^{-12}$ ($7.7 \cdot 10^{-16}$). For δ=0.5ms, arbitrary precision arithmetic would be needed

[11] Notice that we are measuring queue size in packets and considering a reference average packet size, so that larger queues can also be due to bursts of small packets, which we have not investigated in this study.

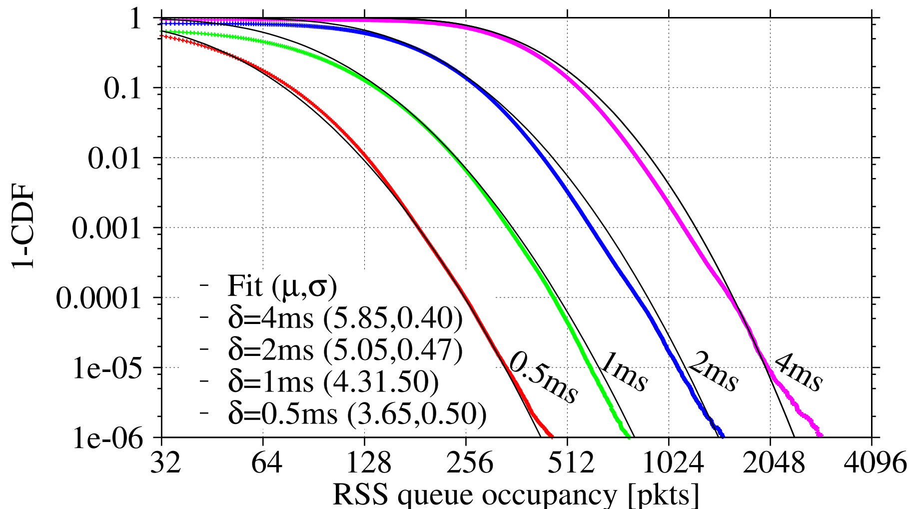

Figure 6: Fitting the tail of the RSS queue occupancy with a lognormal distribution, to estimate RSS overflow probability for a given SD timer δ (sut-NUMA with ISP-80).

to even estimate the overflow probability, that for practical purposes can therefore be considered negligible.

Summarizing, RSS queues of 4096 packets periodically accessed every 0.5ms ensures (i) a sub-ms timestamping precision, (ii) a negligible packer reordering, (iii) extremely unlikely packet losses and (iv) free CPU resources with respect to polling.

5.2 Bounding packet processing time

Processing time outliers. Looking at DPDKStat from the perspective of queuing theory, a strict requirement for system stability is that consumers must be *on average* faster than producers. However, lossless traffic monitoring also requires bounding the duration of *outliers* events. For the sake of illustration, we report in Fig. 7 a sample of packet processing duration. The picture reports temporal evolution, corresponding to 10^6 samples t_i, depicting both the $\hat{t}_i = \alpha t_i + (1-\alpha)\hat{t}_{i-1}$ exponentially weighted moving average (EWMA), as well as the 10^3 samples exceeding the 99.9th quantile.

It can be seen most of the samples exceeding the 99.9th quantile are dispersed in an area just exceeding the quantile (and generate stochastic noise in the EWMA). Yet, very few (and periodic) outliers appears, that have execution times *several orders of magnitude larger* than the average (or even the 99.9th quantile) and that are clearly visible as EWMA spikes. Going back to the queuing interpretation, during an anomalous service time of the consumer, the producer still offer packets, moving them from the RSS queues to the intermediate buffer. As packets pile up unprocessed, this potentially lead to losses: notice that the arrival rate at the RSS queue not only depends on the global arrival rate at the node, but as well on the traffic imbalance (consider the unlucky but realistic case where multiple heavy-hitter flows are hashed to the same RSS queue during such an outlier event). These outliers have a particularly severe effect since, during such time, packet loss can happen.

Garbage collection. Periodicity in the outlier signal is tied to a specific function, namely garbage collection (GC): to avoid locking issues, GC is not implemented as a separate thread, but a function called from the `processPacket()` function (whose statistics are reported in Fig. 7). While it is important to eliminate the occurrence of such ouliers, implementing GC as a separate

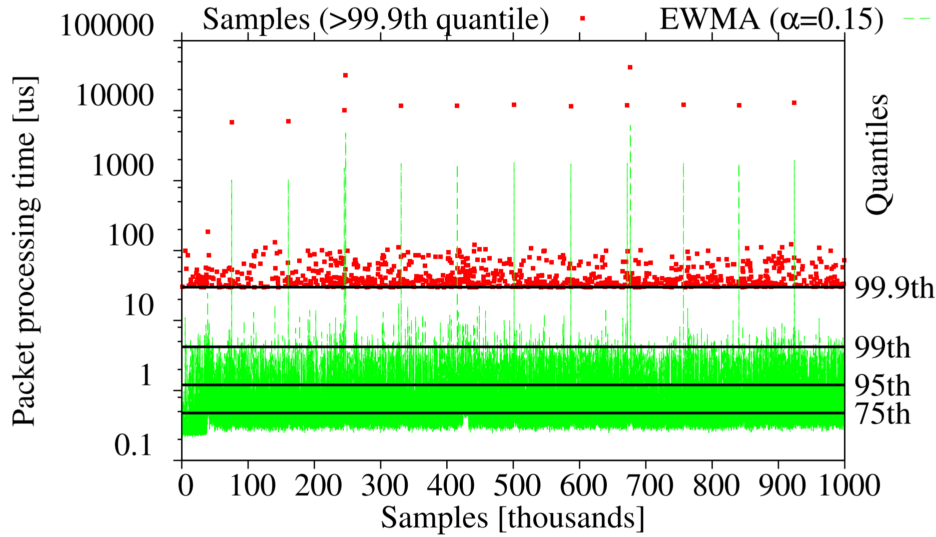

Figure 7: Temporal evolution of packet processing time (10^6 samples): Line represents EWMA ($\alpha = 0.15$) over time, points are the 10^3 samples exceeding the 99.9th quantile (sut-NUMA with ISP-80).

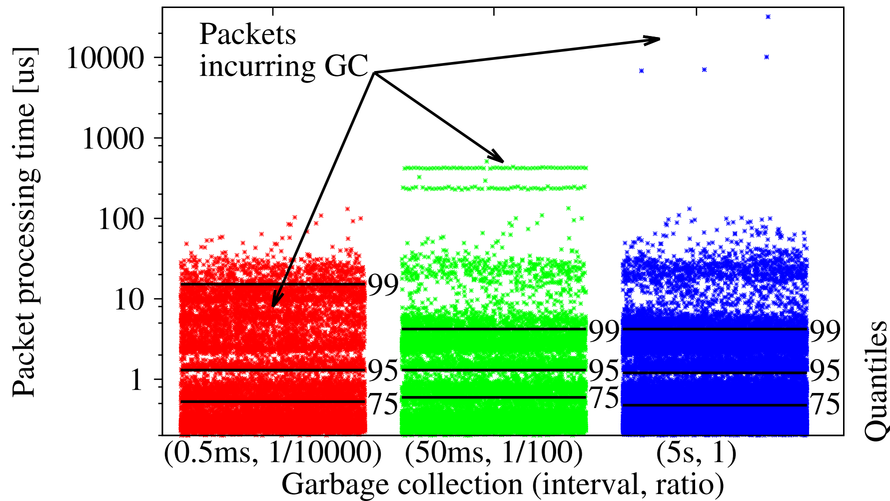

Figure 8: Per-packet processing time for various settings of the garbage collection period and size (sut-NUMA with ISP-80).

thread would not be a concrete solution. Instead, a simple yet effective way to do this is to spread the GC effect over multiple packets: specifically, we fire garbage collection periodically and then process a portion of the full reuse list. Denoting with (period,fraction) the GC settings, Fig. 8 shows the original Tstat setting (5s,1) that scan the full reuse list every 5 seconds, and two additional settings where both the period and the fraction are divided by the same factor: namely, $100\times$ in the (50ms,1/100) case and 10000 in the (0.5ms,1/10000) case. The plot reports horizontal reference lines for 75th, 95th and 99th percentile statistics computed over 10^6 samples (of which we visualize just a portion of the samples shown early in Fig. 7).

Interesting observations hold for Fig. 8: first, notice that the 75th and 95th percentile statistics are indistinguishable across GC settings, which happens since the bulk of the processed packets do not sample a GC event. Conversely, notice that packet processing times affected by a GC event are fairly easily recognizable for the (5s,1) and (50ms,1/100) cases, as they are separate from the bulk of the packet processing time, so that the distribution exhibit a multi-modal behavior. Comparing (5s,1) to (50ms,1/100) we see that as expected outliers become more numerous (by a factor of 100) but their intensity reduces as well (roughly by the same amount). Finally, observe that outliers disappear for (0.5ms, 1/10000), which happens since the portion of the list to be processed by each GC event is now small enough. Also, observe that the 99th percentile is larger than for the other cases, which happens since the number of packets sampling a GC event is large enough to impact the 99th percentile.

Batching size. It is possible to further reduce the period up to another extreme where the GC operations are performed at each packet. This is however not advisable, as it goes against the very same batching principles that circumvented per-packet bottleneck of standard network stacks. In this case, the per-operation overhead is represented by a function call and context switching, that is instead factored out when batching.

GC represents a long sequential operation performed in an "atomic" fashion in the $(\Delta T, 1)$ strategy (irrespectively of the period ΔT), so that the lesson learned here applies a greater extent. Denote with t_{OP} the duration of the monolithic atomic operation (OP), with t the average duration of the packet processing operations that did not incur in OP, and let α be the fraction of OP that will be performed at a rate $1/\alpha$. By definition, a fraction $(1-\alpha)$ of packets will not sample OP, whereas a fraction α will incur in OP overhead, however reduced by (assuming a linear dependency for simplicity) a factor α with respect to the monolithic scenario: hence, the average processing time will be $\mathbb{E}[t] = (1-\alpha)t + \alpha(t + \alpha t_{OP}) = t + \alpha^2 t_{OP}$.

However, as early argued it is not the average processing time, but rather the longest processing times that lead to lossy scenarios. To select α, a better guidance is to upper bound the ratio between the processing time $t + \alpha t_{OP}$ of packets incurring OP and the packets avoiding OP to some factor L, i.e., imposing $(t+\alpha t_{OP})/t \leq L$ so that by choosing $\alpha \leq (L-1)t/t_{OP}$ it is guaranteed that the fraction α of packets incurring OP will have a processing time at most L times larger than that of the other packets ($1 + \alpha(L-1)$ than the average).

Summarizing, it is advisable to break down long and "atomic" sequential tasks into smaller and more frequent sub-operations, spreading the cost to let the packet processing statistics be as much mono-modal as possible and to avoid outliers.

6 Experimental Results

We finally examine DPDKStat raw processing rates, first gathering conservative performance on a single trace, then extending the analysis to all traces. In particular, we investigate system performance with respect to packets acquisition strategies and hyper-threading (Sec.6.1), multiple CPUs (Sec.6.2) and traces (Sec.6.3).

6.1 Periodic acquisition and hyper-threading

Let us focus on sut-SMP first. Fig 9 shows the maximum achievable processing rate without losses when running a variable number of DPDKStat instances. Results compare *polling* (dashed line) with *periodic* (solid line) packets acquisition policies, the latter implemented via `SCHED_DEADLINE` (SD). Policies have a direct impact on processes-to-core allocation: as sketched on the top part of the figure, when using polling the best performance are obtained when packets acquisition (A) and processing (P) threads run on dedicated cores (either physical or logic), while acquisition and processing threads can share the same core with SD.

The two policies present similar performance up to 2 instance, with a small advantage for polling in the single instance case (as 2 physical cores are used). When using more instances, SD

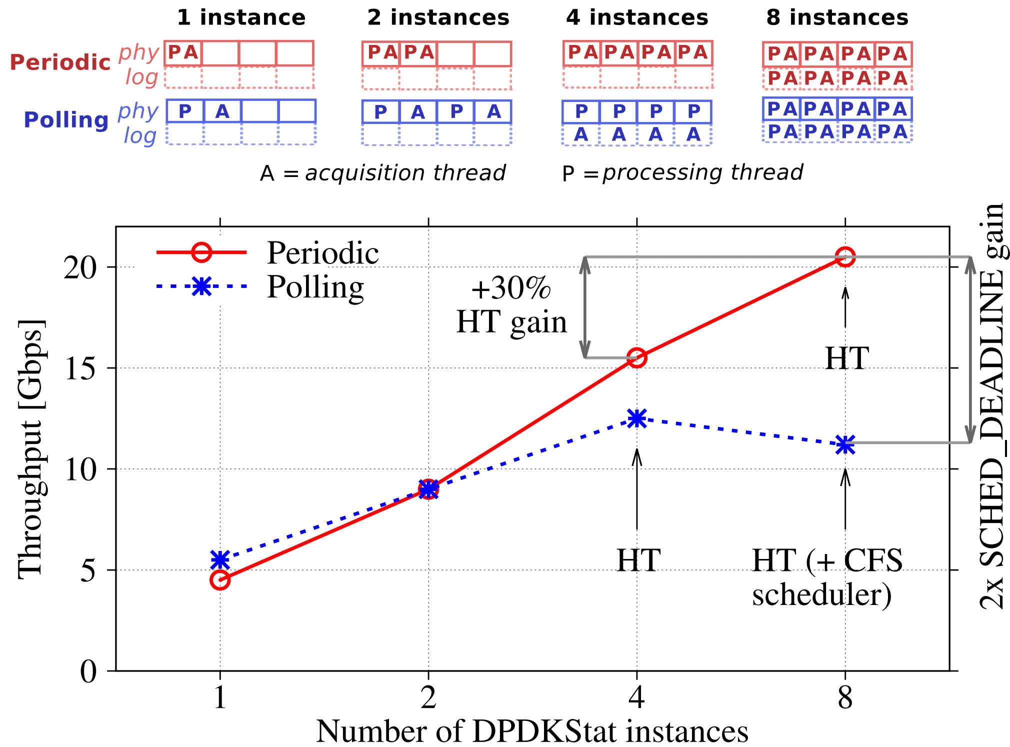

Figure 9: Achievable throughput (sut-SMP with ISP-full).

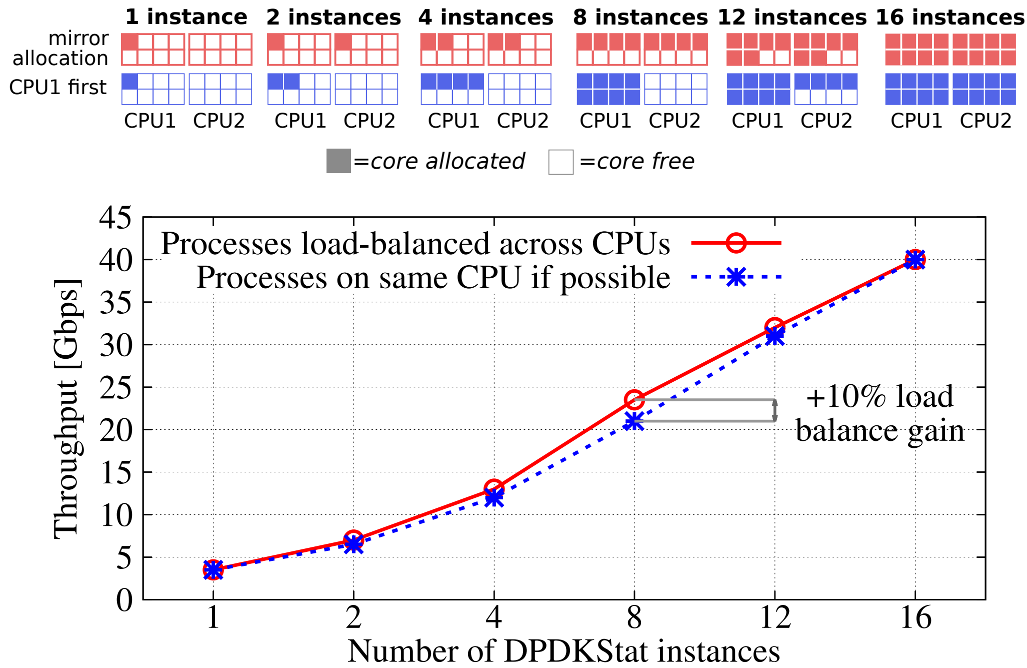

Figure 10: Achievable throughput (sut-NUMA no hyper-threading, with ISP-full).

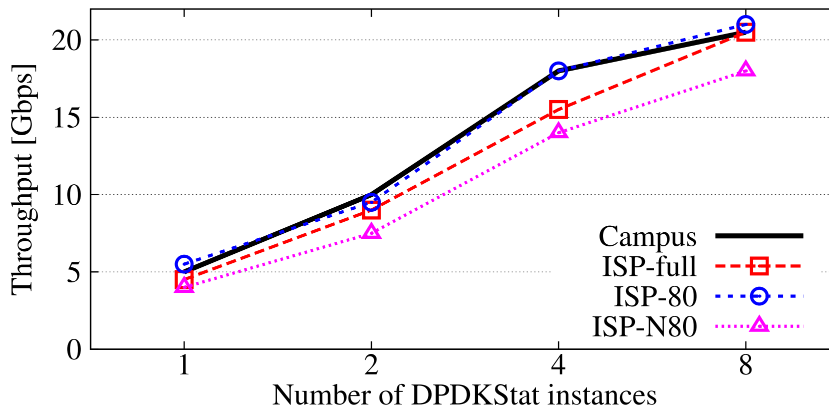

Figure 11: Sensitivity with respect to traffic mixture. (sut-SMP with SCHED_DEADLINE and HT for 4 and 8 DPDKStat instances).

presents large performance improvement with respect to polling, a trend maintained also at full capacity. Overall, the system achieves 21 Gbps throughput without losses, about twice as much as system performance under polling. This is even more impressive considering that the system only has 4 physical cores.

Hyper-threading (HT) possibly yields remarkable performance speed-up: as it can be noted comparing the 4 vs 8 instances under periodic SD acquisition, running twice as many instances in the same amount of silicium yields +30% performance speedup. Conversely, HT gains are more limited under polling: despite hyper-threading yields benefits in the 4 instances scenario, gains are completely offset in the 8 instance scenario due to increased contention – which confirms that polling is not the best strategy for packet acquisition.

It could be objected that few simple "tricks" could be adopted to mimic periodic acquisition directly in the user application, such as introducing controlled `usleep()` to break the packet polling loop (at user-application level) and tweaking the process/thread priorities (in the regular CFP Linux scheduler). Yet, `usleep()` is knowingly unreliable for sub-millisecond sleeps, and may result in longer sleeps (due to system activity and the granularity of system timers), yielding to timestamp imprecisions, reordering and losses. Active sleeps are instead more reliable, but they are equivalent to polling in terms of CPU occupancy. Finally, altering the process/threads priorities in the regular CFP scheduler is non advisable as it would require a fair amount of tuning, with no guarantees. Otherwise stated, and as we previously have observed, whenever a functionality is offered at a lower-level (e.g., RSS vs software load balancing; hardware lock of DPDK buffers vs software mutex; kernel-level scheduling discipline vs poor-man `usleep()`), it is a wise idea to accept the kind offer.

Overall, it is desirable to avoid packet polling due to the unnecessary resource consumption in idle loop, and rather take advantage of SCHED_DEADLINE discipline that jointly yields guarantees on the RSS queue size, as well as free up resources that become available from processing. It is also adviseable to enable hyper-threading, that yields sizeable gains with SCHED_DEADLINE.

6.2 Combining different CPUs

We now consider sut-NUMA, recalling from Fig. 4 that for this system all NICs are directly connected to CPU1. In this scenario, we have an additional degree of freedom in terms of core allocation policies: as schematically represented in top of Fig 10, as we can either (i) preferably use all cores of CPU1, which is closer to the NICs, or (ii) mirror allocation to balance the load across CPUs. The dashed (solid) line in the figure corresponds to the scenarios allocating processes on

the same (different) CPU. In these tests, hyper-threading is disabled and we run all processes on the 16 physical cores only (the maximum supported by the SUT architecture).

As for the previous analysis, throughput scales logarithmically with the numbers of cores, and the system successfully reaches 40Gbps 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 instances 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 line cards are limited to 40Gbps and since (ii) Intel NICs offer a maximum of 16 RRS queues. We can however assess HT gains to hold: in particular, when binding all 16 processes to run only on CPU1 with HT enabled, we achieve 24 Gbps, corresponding to a +20% of performance improvement with respect the 8 instances scenario reported in Fig. 10 (this gain is lower than what obtained from sut-SMP, possibly due to the different HW specs).

Summarizing, load balancing the packet processing over two CPU nodes, including the one far from the NICs, exhibit smaller but still sizeable gain. The overall DPDKStat processing rate could possibly exceed 40Gbps, which we cannot test due to physical limits of our testbed.

6.3 Sensitivity to input workload

We conclude our analysis investigating system performance with respect to different input traffic mixtures. Fig. 11 shows the achievable throughput for sut-SMP using all traces available. Despite results are in line with what reported in Fig. 9, as expected the type of input traffic can have an impact. In particular, the system is more stressed when processing ISP-N80, i.e., the trace composed mostly by P2P traffic. Indeed, in this condition the system need to manage a humongous amount of small UDP flows (e.g., BitTorrent DHT) imposing a non marginal overhead on the data structure and garbage collection. Conversely, it performs the best with Campus (and ISP-80), i.e., the traces composed mostly (exclusively) by HTTP traffic (+16% improvement). ISP-full, being a combination of the traffic types, has performance lying in the middle, testifying that the analysis in the previous section reported conservative performance results, that are typical of the mixture nowadays ISPs offering landline access would incur.

Traffic composition has an impact on the overall performance. For instance, mobile networks traffic is mostly composed of HTTP/HTTPS traffic while fixed access networks present P2P-centric workload. Even if the proposed system can cope with both traffic mixtures, such differences can be further considered to provide specific per-setup tuning and optimizations.

7 Aftermath

We design, implement and benchmark a system for scalable traffic analysis, able to process 40Gbps with common hardware. Our analysis highlights several takeaways. First, periodic packet aquisition policies implemented via the `SCHED_DEADLINE` (SD) kernel policy is very efficient (about a factor of 2 improvement over polling). In reason of the deadline guarantee, the SD technique is also amenable for a precise RSS dimensioning to achieve (stochastically) loss-free operation. Second, hyper-threading (HT) gain is smaller but sizeable (20%-30%). Third, load balancing of processes over multiple NUMA nodes bring a non-neglibile payoff (10%). Fourth, applications must leverage large intermediate buffers to cope with variable processing times, to avoid packet overwrite in the circular buffers. These variable delays can be due to either (i) long tasks of the very-same application, in which case it be preferable to break them down into smaller subtasks; (ii) I/O bottlenecks, in which case a background thread would solve the starvation; (iii) external applications (including system services), that fall outside the control of the application under test, and that while are unavoidable, are also well absorbed by a large buffer. This work also opens a number of interesting points, that we next outline.

Sorting packets. We believe that it would is possible to further increase performance by tweaking the packet arrival process and reordering packets before handing them over the packet analyzer.

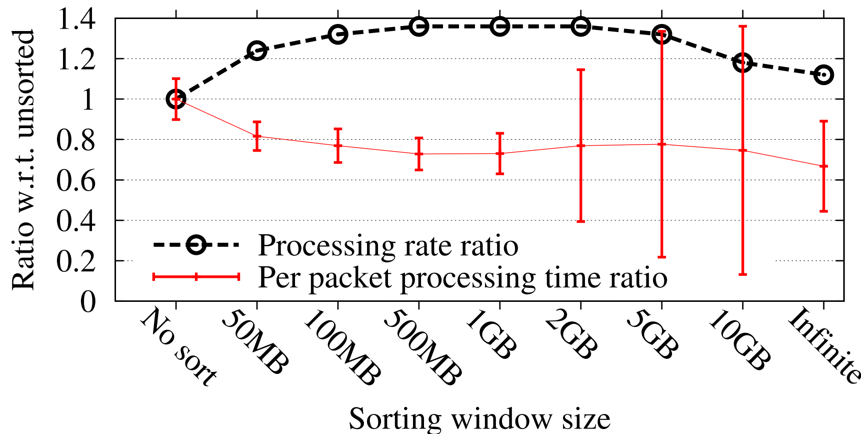

Figure 12: Upper-bound of sorting gains (sut-SMPwith ISP-full).

This would maximize the cache hit ratio related to flows data structure at the cost of more complex operations for the packets acquisition thread.

To preliminary investigate the magnitude of the potential gain, we *pre-sort the traces offline*, so that we gather an upper-bound of the performance gain (as the RSS thread does not have to perform sorting in this experiments). We further consider different horizon of the sorting up to an infinite window (i.e., all the trace is sorted, which is of course unpractical and only included for reference), and report performance gain in Fig. 12. The picture show a non marginal gain, that already appear at around 50MB (about 10 times the size of our RSS queue) and tops at 1GB (the size of our intermediate buffer), after which the gain decreases.

Clearly, gains are tied to the amount of statistical flow multiplexing that windows of a given size can see, so that especially for small window sizes, increasing the window increase the hit ratio. When the window is instead too big, we observe a counter intuivite price penalty, that is however easy to understand by considering the limit case of the infinite window: in this case, the packet analyzer will alternate between (i) periods where data acquisition offers them large elephant flows, benefiting from many cache hits, to (i) periods with more mice than in the regular traffic (since the packets from the elephant flows have been batched together) suffering from many cache misses. These opposite effects also yield to larger variance of the processing rates.

At the same time, notice that to maximize benefits relatively large buffer size need to be processed, so that ideally sorting should happen at the packet analyzer, dequeuing packets from the large buffer according to flow affinity. Interestingly, this would equal to a Longest Queue First (LQF) scheduling policy in the packet analyzer, which is practically implementable and whose stability properties are also well understood [7].

GPU offloading. We have previously argued that GPU are not cost-effective for the kind of operation that STA perform, and that they are doomed for STA in reason of a known bottleneck in moving packet payload [34]. Yet, we have also seen that offloading operations to hardware whenever possible (e.g., RSS, Hyper-threading and software transactional memory using hardware locks) or even to lower-layer sofware (e.g., kernel `SCHED_DEADLINE` vs user `usleep()`) is useful in releaving bottlenecks. Hence, when adding more STA functions to the point where CPU becomes a bottleneck, it could become useful to offload part of the processing to the GPU, with a technique complementary to the one described above. An example is represented by CryptoPAN[12]: this library, currently integrated in DPDKStat, is used to obfuscate IP addresses and only require moving IP addresses as opposite to full payload. A more systematic analysis would be however needed to understand which function of the STA pipelined workflow is appealing to offload to GPUs.

[12] `http://www.cc.gatech.edu/computing/Telecomm/projects/cryptopan/`

Acknowledgements

This work has been carried out during the MSc internship of Martino Trevisan at LINCS http://www.lincs.fr. We are grateful to Fulvio Risso for letting us (ab)use his heavy metal (i.e., sut-NUMA of Fig. 4). The research leading to these results has received funding from the European Union under the FP7 Grant Agreement n. 318627 (Integrated Project "mPlane").

References

[1] http://www.nngroup.com/articles/law-of-bandwidth/.

[2] Tstat-DPDK. http://goo.gl/55mrfP.

[3] Special report on 50 Years of Moore's Law. *Spectrum, IEEE*, 52(4):26–44, 2015.

[4] Actis Dana, A. Progettazione e implementazione di sistema di monitoraggio passivo per reti campus, *MSc Thesis (in Italian)*. http://www.retitlc.polito.it/finamore/papers/Tesi_ACTIS.pdf.

[5] T. Barbette, C. Soldani, and L. Mathy. Fast Userspace Packet Processing. In *ACM/IEEE ANCS*, 2015.

[6] X. Chen, Y. Wu, L. Xu, Y. Xue, and J. Li. Para-Snort: A Multi-thread Snort on Multi-core IA Platform. In *PDCS*, 2009.

[7] A. Dimakis and J. Walrand. Sufficient conditions for stability of longest-queue-first scheduling: Second-order properties using fluid limits. *Advances in Applied Probability*, 38(2):pp. 505–521.

[8] Finamore, A. and Mellia, M. and Meo, M. and Munafò, M. and Rossi, D. Experiences of Internet Traffic Monitoring with Tstat. *Network, IEEE*, 25:8–14, 2011.

[9] F. Fusco and L. Deri. High Speed Network Traffic Analysis with Commodity Multi-core Systems. In *ACM IMC*, 2010.

[10] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. Comparison of Frameworks for High-Performance Packet IO. In *ACM/IEEE ANCS*, 2015.

[11] B. Giuseppe, B. Marco, P. Giulio, P. Salvatore, and M. Marco. StreaMon: A Software-defined Monitoring Platform. In *ITC*, 2014.

[12] Intel. DPDK - Data Plane Development Kit. http://dpdk.org, 2011.

[13] Intel. Delivering 160Gbps DPI Performance on the Intel Xeon Processor E5-2600 Series using HyperScan. http://goo.gl/zVc4xe, 2013.

[14] Intel. Supra-linear Packet Processing Performance with Intel Multi-core Processors. http://goo.gl/MvDszb, 2013.

[15] D. J. Day and B. M. Burns. A Performance Analysis of Snort and Suricata Network Intrusion Detection and Prevention Engines. In *ICDS*, 2011.

[16] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: A Highly-scalable Software-based Intrusion Detection System. In *ACM CCS*, 2012.

[17] H. Jiang, G. Xie, and K. Salamatian. Load Balancing by Ruleset Partition for Parallel IDS on Multi-Core Processors. In *ICCCN*, 2013.

[18] H. Jiang, G. Zhang, G. Xie, K. Salamatian, and L. Mathy. Scalable High-performance Parallel Design for Network Intrusion Detection Systems on Many-core Processors. In *ACM/IEEE ANCS*, 2013.

[19] L. Jun, L. Feng, and A. Nirwan. Monitoring and analyzing big traffic data of a large-scale cellular network with Hadoop. *IEEE Network*, 28(4):32–39, 2014.

[20] L. Koromilas, G. Vasiliadis, I. Manousakis, and S. Ioannidis. Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures. In *ACM/IEEE ANCS*, 2014.

[21] L. Koromilas, G. Vasiliadis, I. Manousakis, and S. Ioannidis. Efficient software packet processing on heterogeneous and asymmetric hardware architectures. In *ACM/IEEE ANCS*, 2014.

[22] D. Luca, M. Maurizio, B. Tomasz, and C. Alfredo. nDPI: Open-source High-speed Deep Packet Inspection. In *TRAC*, 2014.

[23] V. Moreno, P. del Rio, J. Ramos, J. Garnica, and J. Garcia-Dorado. Batch to the Future: Analyzing Timestamp Accuracy of High-Performance Packet I/O Engines. *IEEE Communications Letters*, 16(11):1888–1891, November 2012.

[24] J. Nam, M. Jamshed, B. Choi, D. Han, and K. Park. Scaling the Performance of Network Intrusion Detection with Many-core Processors. In *ACM/IEEE ANCS*, 2015.

[25] K. NamUk, C. GanHo, and C. JaeHyeong. A Scalable Carrier-Grade DPI System Architecture Using Synchronization of Flow Information. *IEEE Journal on Selected Areas in Communications*, 32(10):1834–1848, Oct 2014.

[26] D. Patterson. The trouble with multi-core. *Spectrum, IEEE*, 47(7):28–32, 2010.

[27] Procera Networks. Virtual PacketLogic Software Performance Test. `http://goo.gl/HFJyXz`, 2014.

[28] Sandvine. Policy Traffic Switch: Overview. `https://goo.gl/qiRogN`, 2015.

[29] SANS Institute. Open Source IDS High Performance Shootout. `http://goo.gl/RcneUW`, 2015.

[30] P. M. Santiago del Río, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli, and J. Aracil. Wire-speed statistical classification of network traffic on commodity hardware. In *ACM IMC*, 2012.

[31] D. Simoncelli, M. Dusi, F. Gringoli, and S. Niccolini. Stream-monitoring with Blockmon: Convergence of Network Measurements and Data Analytics Platforms. *SIGCOMM Comput. Commun. Rev.*, 43(2).

[32] R. Sommer, M. Vallentin, L. De Carli, and V. Paxson. Hilti: An abstract execution environment for deep, stateful network traffic analysis. In *ACM IMC*, 2014.

[33] Trevisan, M. DPDK tools. `https://github.com/marty90`.

[34] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: a multi-parallel intrusion detection architecture. In *ACM CCS*, 2011.

[35] Woo, S., Park, K. Scalable TCP session monitoring with Symmetric Receive-Side Scaling, 2012.