Testing the capacity of off-the-self systems to store 10GbE traffic

Victor Moreno, Javier Ramos, José Luis García-Dorado,
Ivan Gonzalez, Francisco J. Gomez-Arribas, Javier Aracil
High Performance Computing and Networking research group,
Universidad Autónoma de Madrid, Spain.

NOTE: This is a version of an unedited manuscript that was accepted for publication. Please, cite as:
Victor Moreno, Javier Ramos, José Luis García-Dorado, Ivan Gonzalez, Francisco J. Gomez-Arribas, Javier Aracil. Testing the capacity of off-the-self systems to store 10GbE traffic. IEEE Communications Magazine, Network Testing Series, Vol. 53, No. 9, pp. 118-125 Sep 2015.
The final publication is available at:
`http://dx.doi.org/10.1109/MCOM.2015.7263355`

Abstract

The maturity of the telecommunications market and the fact that user demands increase every day leaves network operators no option but to deploy high-speed infrastructures and test them in an efficient and economical manner. A common approach to this problem has been the storage of network traffic samples for analysis and replay using different versions of what we have named Network Traffic Storage Systems (NTSS). This type of task is particularly demanding in 10 Gb Ethernet links and is traditionally addressed by closed solutions or NTSS built on top of high-end hardware. However, these approaches lack flexibility and extensibility which typically translates into higher cost. This work studies how NTSS can be built using commodity off-the-shelf systems (COTS): a combination of commodity hardware and open source software. To this end, we present the current limitations of COTS systems and focus on low-level optimization techniques at several levels: the NIC driver, hard drives, and the software interaction between them. The application of these techniques has proven crucial for reaching 10 Gb/s rates as different state-of-the-art systems have shown after an extensive performance test.

Keywords: Packet sniffing; packet storage; off-the-shelf systems; 10 Gb Ethernet; NTSS.

1 Introduction

Operators are currently deploying novel network architectures and equipment with bandwidth capabilities of multi-gigabit rates and beyond. Testing the performance and correct operation of such deployments is a challenging task that operators must face. This also applies to other players in the Internet arena such as companies, third-party enterprises, and banks that deploy new services for their customers and employees.

The most simple and efficient way of testing such infrastructures and services is sniffing and storing all the traversing test traffic for its subsequent analysis [1]. Such an analysis may focus not only on searching malformed or unexpected packets (e.g., erroneous VLAN or MPLS headers, or duplicated frames) but also on the network performance and Quality of Service (QoS) parameters' value—bandwidth, packet loss, delay or jitter. Additionally, stored traffic may be used not only passively but also actively when replaying the content of the stored traces for testing purposes [2]. We suggest referring to those systems that sniff and store traffic as Network Traffic Storage Systems (NTSS).

Even an intuitively simple task such as sniffing and storing the traversing traffic is a challenge when dealing with 10 Gb/s rates or higher due to the great amount of resources and computational power needed. Traditionally, specialized hardware devices such as FPGAs, network processors, and high-end closed commercial solutions have been applied to tackle the traffic sniffing and storage problem. Such solutions address the performance part of the problem in a very effective way, and they also offer high degrees of both determinism and robustness, desirable for any industrial development. However, such positive features are obtained at the expense of flexibility and

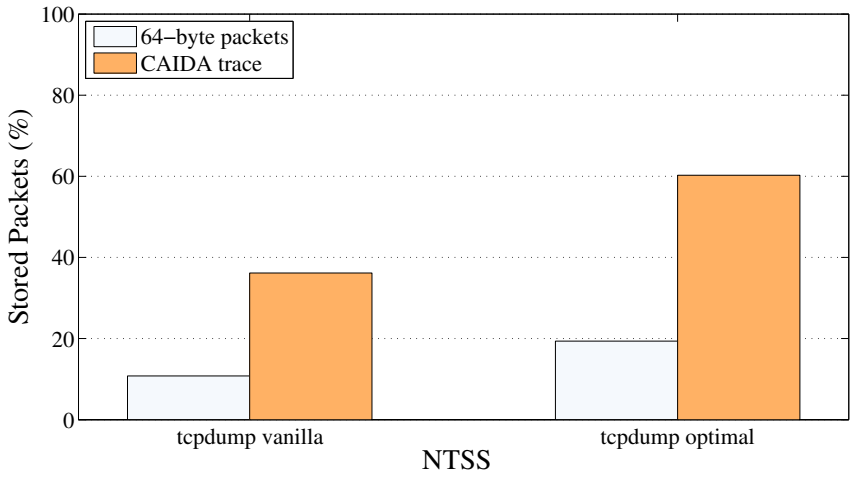

Figure 1: Percentage of stored packets (into a RAID-0 volume with 9 disks) for a full-saturated 10 Gb Ethernet link for NTSS based on tcpdump with a vanilla and optimized configuration for a 30-minute experiment

extensibility, which turns the deployment, evolution, and maintenance processes into difficult tasks. Furthermore, the prices of those systems is elevated, ranging from $85,000 to $250,000 depending on their storage capacity[1,2].

As an alternative, the research community has recently focused on commodity-off-the-shelf (COTS) solutions to accomplish high-performance tasks [3]. COTS systems have emerged as the combination of commodity hardware and open source software. Such systems provide flexibility, availability, and scalability while handling multi-gigabit rates and cutting expenditures both in terms of deployment and maintenance [4]. For example, the system used for the experiments presented in this article had a price of $10,000, which is one order of magnitude lower than the price of its closed solution counterpart. With this in mind, this article explains the key aspects for COTS systems to sniff and store packets at multi-gigabit rates. Specifically, such keys comprise fine low-level tuning at NIC driver, hard drives, and application levels. Subsequently, we provide an extensive performance evaluation of state-of-the-art NTSS systems that have successfully reached such a goal.

2 Problem statement and contributions

Traffic storage has become a challenging task, as a fully-saturated 10 Gigabit Ethernet (GbE) link in the worst-case scenario (minimal size packets, i.e., 64 Bytes on Ethernet with CRC included) carries more than 14 million packets per second. In this demanding scenario, we first note how much traffic may be sniffed with standard software running on a commodity server. Specifically, our commodity server is a Supermicro X9DR3-F with two 6-core Xeon E5-2630 processors running at 2.30 GHz and hyperthreading disabled, with 96 GB of DDR3 RAM at 1333 MHz. The server is equipped with an Intel 82599 10 GbE NIC plugged into a Peripheral Component Interconnect Express (PCIe) 3.0 slot. The software is composed of an Ubuntu Server 14.04 configured with a 3.14 kernel and the default network stack, the vanilla Intel `ixgbe` NIC driver, and the de-facto standard traffic sniffer `tcpdump`. In the sender side, we have an FPGA-based traffic transmission system capable of replaying at link-rate both fixed-size synthetic traffic and packet traces previously stored in the machine [5].

The results are shown on the leftmost 2-column group in Fig. 1, which show the percentage of stored packets with this configuration (named `tcpdump` vanilla), for two traffic injection cases: namely, synthetic 64-byte packets (CRC included) and a real backbone trace from CAIDA[3], both replayed at wire-speed. We observe that this out-of-the-box scenario can only sniff and store less than 10% and 38% out of the total sent packets, for synthetic and real traffic, respectively. The out-of-the-box configuration has thus proven insufficient to capture full-rate 10 GbE traffic.

Nevertheless, the research community has first tried to improve the NIC vanilla driver to boost up its performance, as reviewed in the following section, but there is scarce knowledge on how to write such traffic in hard drives at multi-gigabit rates. Consequently, this article studies how to tune such drives to increase their performance, a question that is dealt in Section 4. Then, Section 5 explains how to optimally combine sniffing and storing, and provides a performance evaluation of the state-of-the-art NTSS in 10 GbE networks. Finally, the guidelines and take-away messages presented along this paper are summarized in Section 6 together with the proposed future work on the field.

[1] http://www.netapp.com/products/storage-systems/ef-series/
[2] http://www.napatech.com/products/
[3] http://www.caida.org/data/passive/passive_2009_dataset.xml

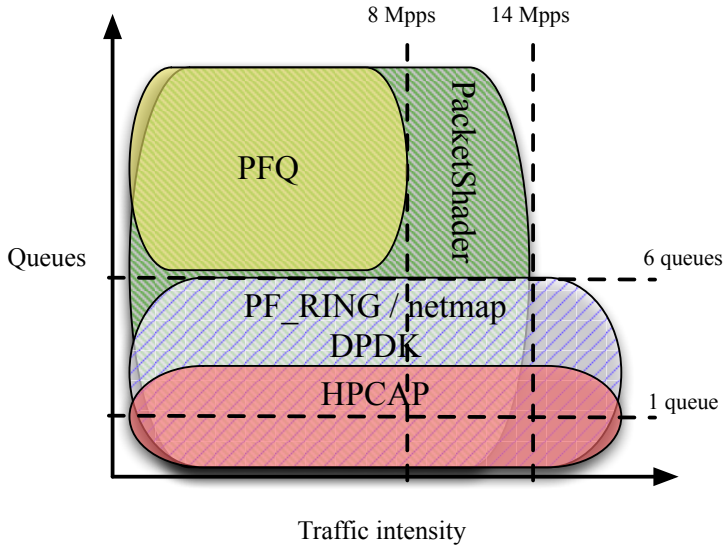

Figure 2: Qualitative comparison between the existing High-Performance packet sniffing engines

3 Sniffing traffic from the wire

The first task is to sniff the traffic from the wire. Alternatively this task was traditionally termed 'capture'; however we will avoid the term here as it is easily confused with the term 'store'. The following performance optimizations techniques have been applied to the sniffing process in the literature:

- *Pre-allocation and re-use of memory*: On vanilla drivers, for each received packet, a set of structures and buffers is allocated. Those resources are released once the packet is delivered to upper layers. It has proven more efficient to pre-allocate a pool of structures and re-use them for subsequent incoming packets.
- *Memory mapping*: The use of these techniques (such as Direct NIC Access, DNA [6]) allows high-level applications to map the receiving buffers located at driver-level, thus reducing the number of copies.
- *Use of parallel direct paths*: Modifying the network driver to bypass the operating system's network stack makes it possible to create parallel paths from the NIC (Receive Side Scaling, RSS) to user-level applications. As collateral effects, more CPU cores are necessary and packet reordering may occur [7].
- *Batch processing*: Typically, upper-layer applications make a system call to receive a packet. Processing a batch of packets per system call can reduce the resulting overhead. Nevertheless, such techniques may also entail a latency increment and inaccurate timestamping [8].
- *Prefetching*: This technique consists of pre-loading memory locations in processors' caches in a predictive way so that they can be quickly accessed in the near future should they be needed, thus reducing cache misses.
- *Affinity*: Non-Uniform Memory Access (NUMA) architectures group processing cores along with independent memory banks to create a NUMA node. Performance is significantly increased if the sniffing processes is placed on the same NUMA node as the driver-level receiving threads, thus reducing the data access overhead [4].

We note that the techniques to improve sniffing performance require changes either in the NIC driver or in the operating system's network stack, or in both. Nevertheless, some straightforward affinity-related adjustments may be performed in an out-of-the-box Linux machine with `tcpdump`, with significant performance improvements as shown in Fig. 1 labeled as `tcpdump` optimal. The main difference between the vanilla and the optimized execution of `tcpdump` lies in the CPU assignment of the sniffing process and the number of reception queues. In the vanilla case, the program runs in a random processor core selected by the operating system and the number of reception queues is set to the total number of cores of the machine—12 in our case. In the optimized case, the capture process runs in the NUMA node attached to the NIC with 4 reception queues whose interrupts were also attached to that node. This 4-queue configuration showed the maximum throughput in our setup. With this

Table 1: Maximum rates generated by a fully-saturated 10GbE link in terms of packets and bits for different packet sizes and header formats

Max. throughput	**Packet size (bytes, CRC included)**								
	60	**64**	**128**	**256**	**512**	**750**	**1024**	**1250**	**1514**
Mp/s	14.88	14.21	8.22	4.46	2.33	1.62	1.19	0.98	0.82
Gb/s	7.14	7.27	8.38	9.13	9.55	9.69	9.77	9.81	9.84
Gb/s (PCAP header included)	9.05	9.09	9.46	9.71	9.85	9.90	9.92	9.94	9.95
Gb/s (RAW header included)	8.57	8.64	9.19	9.57	9.77	9.84	9.89	9.91	9.92

simple change, the number of stored packets increases up to 20% and 60% (from 10% and 38%), for synthetic and real traffic, respectively.

To fully sniff high-speed traffic, some engines were proposed [9, 10, 11, 12, 13][4], which apply the aforementioned techniques over the driver and the network stack. Figure 2 shows a qualitative comparison between the most popular capture engines in terms of traffic intensity and number of receive queues needed to work. Specifically, PacketShader, PF_RING, netmap, and DPDK achieve wire-speed using one receive queue. `HPCAP` obtains similar results but it additionally provides accurate timestamping by using two CPU cores per receive queue, while the other engines use one core per queue. PFQ needs a larger number of queues and cores to give more flexibility and additional functionalities, such as customized packet aggregation, at the expense of performance. The reader is referred to [4], for further details.

4 Storing data on hard-drives

According to manufacturers' specifications, a high-end SATA-3 mechanical disk allows theoretical rates up to 4.8 Gb/s for sequential reads and 1.2 Gb/s for sequential writes. A SATA-3 Solid-State Drive (SSD) may achieve speeds close to 3.2 Gb/s for both read and write operations, but the price per GB is 10 times greater. Consequently, no matter if the hard drive is SSD or conventional, a single disk is not enough to comply with the line rate of 10 GbE networks and a RAID volume is in order.

In terms of packet sniffing the worst case scenario is when packets are of minimal size (64 bytes, CRC included, for 10 GbE) and, therefore, the quantity of packets per second to be processed is maximized—14.88 Mp/s. However, we note that the worst case for traffic storage is the opposite (maximum sized packets, i.e., 1514 for 10 GbE leading to 816 Kp/s), as shown in Table 1. It turns out that the disc load increases with packet size and, to complicate matters, an additional header is aggregated to the packet with the timestamp and both packet and capture lengths. For instance, the de-facto PCAP standard stores a 16-byte header per packet—4 bytes for caplen, 4 bytes for len and 8 bytes for timestamp. Other packet storage formats such as the reduced PCAP (labeled as RAW) reduces by 2 the amount of bytes required for each length field. Table 1 shows the storage overhead using PCAP and RAW formats.

Table 1 shows that the requested disk capacities are larger than 9 Gb/s with a worst case of 9.95 Gb/s assuming PCAP headers. As shown, even for the mean Internet's packet size (i.e. ranging between 256 and 512 bytes according to CAIDA), the demand for store throughput ranges between 9.71 and 9.85 Gb/s, respectively. As a conclusion, once the sniffing engines proved capable of dealing fairly with all packet sizes, now the goal is for the RAID volume to attain rates of nearly 10 Gb/s.

Unfortunately, the amount of parameters involved in RAID configuration is large and a wrong choice of values may lead to severe performance degradation. We have evaluated the actual write throughput of a RAID-0 volume composed of high-end mechanical hard disks (Hitachi HUA723030ALA640 with 3TB of capacity) or SSD (Samsung 840 EVO with 250 GB of capacity) both with SATA-3 interfaces, using the out-of-the-box Linux server described in Section 2. Specifically, we have conducted thorough testing with the following configuration parameters:

- *Number of disks*: We have assessed how performance varies with number of disks ranging from 1 to 12 for mechanical drives and from 1 to 8 for SSD, merged into a RAID-0 volume—we used an Intel RS25DB080 RAID controller.
- *Strip size*: The strip size is the amount of data per basic write operation. Thus, small strip sizes will be translated into a higher number of write operations into the RAID volume and may degrade the overall

[4]Intel Data Plane Development Kit (Intel DPDK) Release Notes are in `http://www.intel.com/content/dam/www/public/us/en/documents/release-notes/intel-dpdk-release-notes.pdf`

Table 2: Write throughput summary results

Number of disks	Technology	Scenario	Strip size	RAID cache policy	Disks' cache	FS	Throughput (Gb/s)			
							average	confidence interval ($\alpha = 0.01$)	Percentile 5^{th}	Percentile 95^{th}
1	Mech	min	1 MB	WTC	off	jfs	0.69	$(0.67, 0.70)$	0.58	0.77
		max	64 kB	WBC	on	xfs	1.27	$(1.26, 1.27)$	1.26	1.27
	SSD	min	64 KB	WTC	off	xfs	0.58	$(0.58, 0.59)$	0.55	0.62
		max	1 MB	WBC	on	jfs	2.21	$(2.18, 2.23)$	2.19	2.21
8	Mech	min	64 kB	Direct	off	jfs	3.64	$(3.51, 3.76)$	2.90	4.19
		max	1 MB	WBC	on	xfs	10.06	$(10.01, 10.12)$	9.77	10.35
	SSD	min	64 KB	WTC	off	jfs	2.15	$(1.99, 2.31)$	1.68	3.82
		max	1 MB	WBC	on	xfs	7.52	$(6.54, 8.51)$	3.17	15.97
9	Mech	min	1 MB	Direct	off	jfs	4.14	$(3.98, 4.30)$	3.27	4.81
		max	1 MB	WBC	on	xfs	11.31	$(11.25, 11.37)$	10.96	11.47

write throughput due to per-operation overheads. We have evaluated strip sizes of 64 KB, 256 KB and 1MB.

- *RAID write cache policy*: This parameter refers to the use of the RAID controller's cache memory. The *D*irect policy disables the cache and performs poorly. The *Write Through Cache* (WTC) policy writes the cache content to disk, and then, a new cache write operation proceeds. Thus, when using WTC the data has to be stored both into the disks and their caches before a new write operation is started. Finally, the *Write Back Cache* (WBC) policy is less conservative and does not require the cache to be flushed to the hard disks before a new write operation is performed.

- *Disk cache*: Some hard drives feature a cache that performs bundling of write operations to a given sector, thus saving disk head movements. We have considered this option in our experiments.

- *Filesystem*: We have evaluated the `ext4` filesystem, which is the de-facto standard for Linux systems. Additionally, we have tested the `xfs` and `jfs` filesystems, as a previous analysis highlighted them as promising candidates. Specifically, `xfs` was designed with the goal of managing a large number of big files. We have additionally tested the RAID's write throughput when no filesystem is instantiated, as a baseline.

The experiments were carried out by taking all possible combinations of the parameters and results are shown in Table 2. For each combination, one hundred 2 GB-sized files were written using the Linux `dd` tool. Specifically, the table depicts which parameter combination offers the minimum and maximum write throughput for different number of disks. For each parameter combination we show the mean write throughput, the confidence interval with a 0.01 significance level and the $5^{th}/95^{th}$ percentiles.

In practical terms, Table 2 shows that a single mechanical disk has an average write throughput of 1.26 Gb/s for its best configuration, with a narrow confidence interval and a percentile range of roughly tenths of Mb/s. Interestingly, average throughputs scale linearly with the number of disks when they are optimally configured, but this linearity is not observed for the worst-performing parameter combinations. Our findings show that 8 disks suffice for all scenarios and packet sizes under study if a properly sized buffer is used to absorb peaks in the write throughput. In fact, the 5^{th} percentile for the throughput obtained with 8 disks, which is 9.77 Gb/s, is below the target for some of the most typical scenarios on the Internet assuming both RAW and PCAP headers.

Additionally, we note that a 9-disk RAID exceeds the target rate both in the mean rate and corresponding 5^{th} percentile. This configuration presents a good trade-off to cope with the oscillations that commodity hard-drives experience, and it can handle all packet storage scenarios for 10 GbE networks even with the largest packet size.

Interestingly, Table 2 also shows that a single SSD drive is capable of consuming nearly twice as much worth of data than a mechanical one—an average of 2.21 Gb/s in contrast with 1.26 Gb/s obtained from the mechanical counterpart. However, our results show that the write throughput scaling ratio (with number of disks) decreases for SSD disks and remains almost constant for mechanical drives. By the time the set of mechanical disks has already achieved an average throughput of 10 Gb/s (i.e., 8 disks), the SSD alternative is far below with the same number of disks: as the table shows it only reaches 7.52 Gb/s. Moreover, the throughput oscillations of the SSD RAIDs were far larger for all filesystems. Specifically for the best case, it was more than an order of magnitude larger than its mechanical alternative—see the width of the confidence interval for the mean. The above issues, together with cost, discourage the use of SSD disks for our packet storage purpose.

Once we have studied how to gauge a RAID volume, we turn our attention to explain how the configuration parameters and their interactions impact performance. To this end, we posed a balanced full-factorial analysis of the data for a RAID-0 array with 9 mechanical hard drives. In such analysis, the response variable under study (in

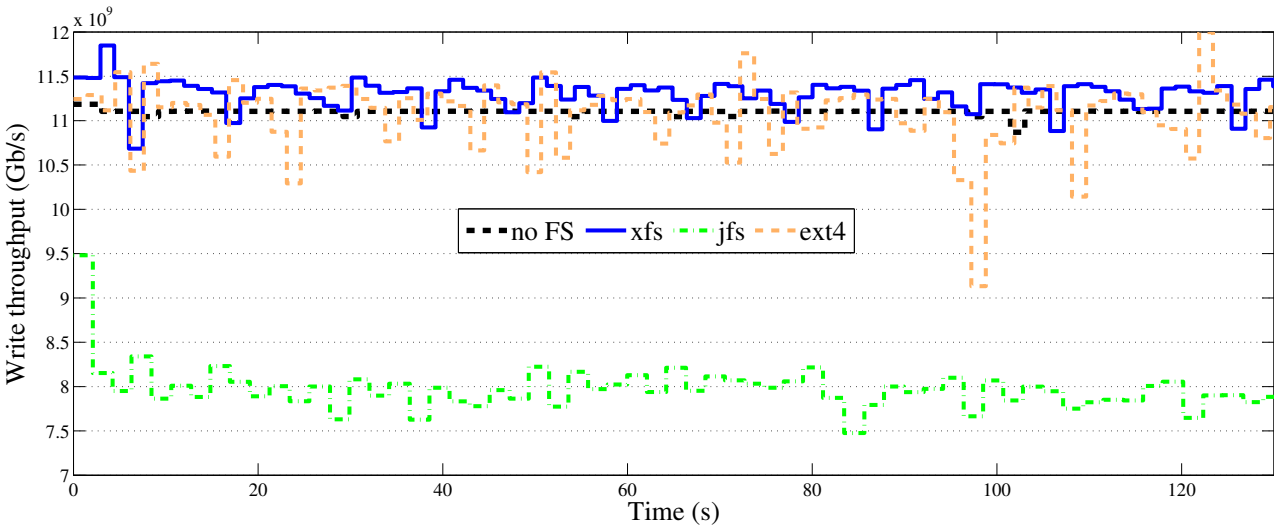

Figure 3: Influence of the filesystem on a 9-disks RAID-0 volume

this case the write throughput) is explained as the outcome of a set of factors (i.e., *Strip size*, *Raid cache policy*, *Disks cache*, and *Filesystem*), and their respective levels—i.e., possible values each factor takes: *Strip size* = 64 KB, 256 KB or 1MB; *Raid cache policy* = Direct, WB or WT; *Disks cache*=off or on; and *Filesystem*= ext4, jfs, or xfs.

All factors and their interactions turned out to be statistically significant in our dataset. Thus, each sample is characterized by the addition of 16 terms: The overall sample mean (often referred as to intercept), one term per main-effect factor that accounts for the different response that each level of such factor gives, and an additional term per each of the possible combinations of factors (six pairs, four trios, and a quartet), which account for the different impact that each combination of levels exerts on the sample.

Several conclusions arise from the analysis, especially the importance of hardware caches. Starting from an overall mean of roughly 4 Gb/s, the use of disk caches represented an average addition of 3.1 Gb/s and similarly, the use of WBC policy gave an average gain of 3.3 Gb/s. The volume's strip size had a relatively marginal significance, which translated into a few hundred of Mb/s for the best configuration—a strip size of 1 MB. On the other hand, the choice of file system was also significant: `xfs` showed the best results with an increase of 0.7 Gb/s in mean compared to `jfs`, that showed the worst results. Finally, the contribution in absolute value of the terms accounting for all possible combinations of levels was limited, from a few tens to one hundred Mb/s.

Furthermore, we found that the filesystem choice does not only affect the average write rate, but also exerts a critical effect on its variance. Fig. 3 shows the throughput obtained when writing the same files as in the previous experiment (one hundred 2 GB-sized files) on a 9-disk RAID-0 volume with the optimal configuration for each filesystem. The figure shows that writing data on the raw volume with no filesystem present has a low-variance behavior. This is a non-practical scenario because data cannot be accessed afterwards, although it is of interest for baselining purposes. When a filesystem is incorporated, throughput oscillations happen, which may be severe. Interestingly, both Fig. 3 and Table 2 show that the `xfs` filesystem presents the smallest oscillation, which makes it the filesystem of choice. For `jfs` and `ext4`, the throughput oscillation may require adding more disks to the RAID volume to ensure that in case of oscillations the RAID meets the target rate.

5 Network traffic storage solutions

Once we have discussed how to optimize both network traffic sniffing and data storage processes separately, we proceed to optimally combine both and come up with a cost-effective high-performance NTSS. More specifically, we outline the fundamental techniques the NTSS may adopt:

- *System call minimization*: The sniffing and storage processes imply data transfer and synchronization between user and kernel level contexts. We seek to minimize the quantity of system calls to reduce context switches and improve overall performance, by using buffer mapping or accessing data in a byte-stream or batch fashion rather than in a per-packet basis.

- *Huge intermediate receive buffers*: As mentioned in Section 4, the target storage device may experience sudden write throughput drops, which can be avoided by means of large intermediate buffers.

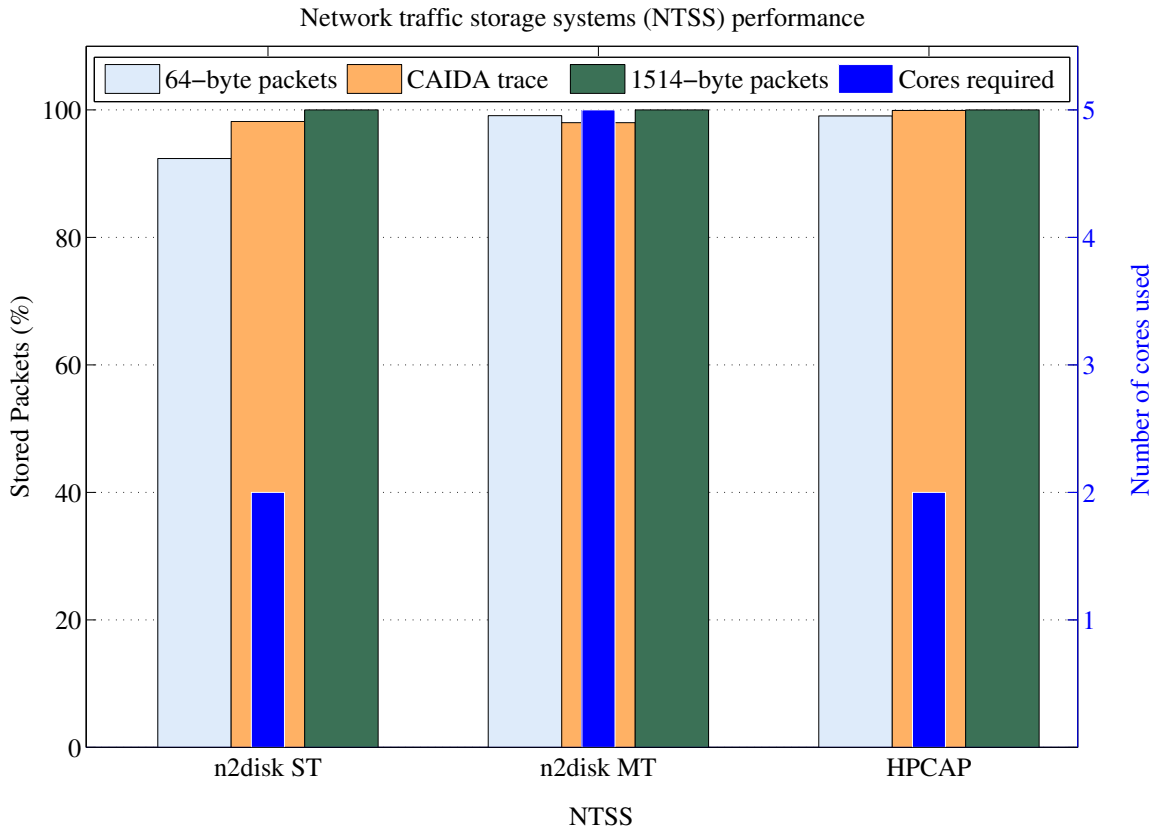

Figure 4: Percentage of stored packets (into a RAID-0 volume) for a full-saturated 10 Gb Ethernet link versus the number of occupied CPU cores for a 30-minutes experiment

- *Memory-alignment*: Maximum write performance is achieved when the transfers between system memory and the storage device are done via Direct Memory Access (DMA) operations (`O_DIRECT` flag in the write options). This way, neither CPU cycles nor cache management nor memory bandwidth is spent in the data transfers. However, this efficient configuration requires the transfer operations to be page-aligned, making memory alignment a critical feature. Moreover, packets must be transferred in blocks with a size multiple of the disk's sector size to reach maximum write performance and avoid quantization effects.
- *Sniffing and storage overlapping*: Overall performance can be increased if the sniffing and storage processes are isolated, allowing their execution to be parallelized.
- *Timestamping*: As explained in [1], accurate packet timestamping is a critical issue, which depends on the low-level NTSS sniffing techniques adopted [8].

Note that only two of the approaches explained in Section 3, PF_RING and `HPCAP`, gave rise to final NTSS, `n2disk` and `hpcapdd`, respectively. On the one hand, `hpcpapdd` was developed on top of the `HPCAP` driver. Both the driver and the application were designed with the goal of optimizing network traffic storage [13]. Regarding the aforementioned techniques, the `HPCAP`+`hpcapdd` system instantiates a 1 GB kernel-level buffer, limited by the kernel configuration. The driver is in charge of timestamping and copying the incoming packets into this buffer, so `hpcapdd` can access them in a byte-stream basis. This buffer is efficiently accessed as it is properly aligned and mapped at user-level. Furthermore, this buffer isolates the sniffing and storage processes, so the overall process is parallelized and pipelined.

On the other hand, `n2disk` has been recently developed by the authors of PF_RING [14]. Specifically, `n2disk` instantiates one or more packet storage threads which are executed in parallel, leading to single-threaded (ST) and multi-threaded (MT) versions. Each thread has an independent memory buffer, and traffic is distributed among them using a hash function. Importantly, `n2disk` not only stores the incoming packets, but also creates additional index files for optimizing subsequent access to the stored data.

For both NTSSs, we measured the percentage of incoming traffic stored versus the number of cores with an optimized 9-disk RAID-0 volume—as discussed in Section 4. Fig. 4 shows the measured percentage of packets stored over the total link load, along with the amount of fully occupied CPU cores used by each NTSS. Those results are shown for the worst-case scenarios in terms of both packet sniffing and storage and in an average case.

Remarkably, `hpcapdd` is capable of storing 99.2% of the incoming traffic for synthetic 64-byte packets (CRC included) and all the real-traffic and synthetic maximum-sized packet traces, with two CPU cores.

The results show that `n2disk`'s single-thread version uses one thread for packet sniffing and one more thread for storage whereas the multi-thread version uses one thread for sniffing and four threads for processing and storing. The ST version stores 92.4% of the packets for the 64-byte experiment, and 98.2% for both maximum-sized packets and real traffic. The MT counterpart stores 99.1% of the packets for the 64-byte experiment, and 98.0% for both maximum-sized packets and real traffic. These last results show that instantiating several threads helps when dealing with the worst-case sniffing scenario, i.e., 64-byte packets, but does not solve demanding storage throughput scenarios.

6 Final remarks and future work

Given the growing importance of COTS systems for network monitoring tasks, this work has addressed the problem of traffic sniffing and storage at 10 Gb/s rates following a bottom-up approach.

First, out-of-the-box tools (i.e., vanilla network drivers and `tcpdump`) have proven to be insufficient to sniff and store packets at 10 Gb/s using off-the-shelf systems. Although the obtained performance is far below 10 Gb/s, the application of some ideas discussed in this work improved the performance of such tools up to rates that may be useful in networks with low utilization.

Second, to improve the sniffing performance and make the most of commodity multi-core servers and modern NICs several optimizations must be applied. For example, affinity planning is a key factor to improve the performance for both optimized and out-of-the box applications. Additionally, some of these optimizations present collateral effects such as timestamp accuracy degradation or packet reordering when applying batch processing and multi-queue reception, respectively. Thus, depending on the application requirements, such optimizations may not apply.

Third, despite a single commodity hard drive is not able to achieve enough write throughput to cope with a 10 Gb Ethernet link, such storage performance may be improved by skillfully using tuned RAID volumes. After this tuning, a write throughput beyond 10 Gb/s is achieved using 9 high-end mechanical disks. The write throughput of commodity hard-drives presents significant oscillations over the mean across time. Such oscillations may be controlled using `xfs` file-system, whereas the use of other file-systems causes remarkable excursions. Such excursions are even more noticeable using SSDs. In fact, although one single SSD achieves more throughput than a rotational drive they do not scale proportionally. Thus, SSDs are left behind their mechanical counterpart.

Finally, one important fact is that the most demanding scenario in terms of packet sniffing (i.e., minimal-size packets) is the least demanding scenario in terms of packet storage throughput. Conversely, the best case for packet sniffing (i.e., maximal-size packets) becomes the most demanding scenario in terms of packet storage throughput. Thus, obtaining maximum performance in a NTSS does not only imply properly tuning the sniffing and storage processes alone, but also their interaction.

As future work, the scaling of the mentioned packet storage solutions to higher link rates must be considered—i.e., 40 Gb/s and 100 Gb/s. Current hardware limitations may prevent a linear scaling of the results shown along this work and more complex techniques will be required. In this line, we propose: (*i*) the study of techniques that allow reducing the amount of information to be stored by smartly selecting the most interesting parts—e.g., the first bytes of each flow or packet [1, 15]; and (*ii*), reducing the magnitude of the stored data by aggregating through some criteria—i.e., moving from packet traces to flow records reduces the storage in one order of magnitude. Needless to say, those techniques come at the expense of potential information losses. On the other hand, the use of a high-performance traffic distribution would open the possibility of replicating standalone systems to achieve the desired rates, although transferring the problem to the distribution system. Anyway, regardless the approach chosen, most of the guidelines presented along this paper still apply.

To conclude, this work has provided both the research community and practitioners with a roadmap not only to understand and use state-of-the-art NTSS systems based on COTS, but also to implement and deploy their own systems. We expect the lessons and ideas we share here may open new opportunities to the use of COTS systems in areas traditionally reserved for high-end and expensive hardware.

References

[1] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, "Enriching network security analysis with time travel," in *Proceedings of ACM SIGCOMM*, 2008, pp. 183–194.

[2] Y.-D. Lin, P.-C. Lin, T.-H. Cheng, I.-W. Chen, and Y.-C. Lai, "Low-storage capture and loss recovery selective replay of real flows," *IEEE Communications Magazine*, vol. 50, no. 4, pp. 114–121, 2012.

[3] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle, "Comparing and improving current packet capturing solutions based on commodity hardware," in *Proceedings of ACM Internet Measurement Conference*, 2010, pp. 206–207.

[4] V. Moreno, J. Ramos, P. M. Santiago del Río, J. L. García-Dorado, F. Gomez-Arribas, and J. Aracil, "Commodity Packet Capture Engines: tutorial, cookbook and applicability," *IEEE Communications Surveys and Tutorials*, to appear.

[5] J. Zazo, M. Forconesi, S. Lopez-Buedo, G. Sutter, and J. Aracil, "TNT10G: A high-accuracy 10 GbE traffic player and recorder for multi-Terabyte traces," in *Proceedings of Conference on Reconfigurable Computing and FPGAs*, 2014, pp. 1–6.

[6] L. Deri, "nCap: wire-speed packet capture and transmission," in *Proceedings of IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, 2005, pp. 47–55.

[7] W. Wu, P. DeMar, and M. Crawford, "Why can some advanced Ethernet NICs cause packet reordering?" *IEEE Communications Letters*, vol. 15, no. 2, pp. 253–255, 2011.

[8] V. Moreno, P. M. Santiago del Río, J. Ramos, J. Garnica, and J. L. García-Dorado, "Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines," *IEEE Communications Letters*, vol. 16, no. 11, pp. 1888–1891, 2012.

[9] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in *Proceedings of ACM SIGCOMM*, 2010, pp. 195–206.

[10] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi-core systems," in *Proceedings of ACM Internet Measurement Conference*, 2010, pp. 218–224.

[11] L. Rizzo, "Revisiting network I/O APIs: the netmap framework," *Communications of the ACM*, vol. 55, no. 3, pp. 45–51, 2012.

[12] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On multi-gigabit packet capturing with multi-core commodity hardware," in *Proceedings of Passive and Active Network Measurement Conference*, 2012, pp. 64–73.

[13] V. Moreno, P. M. Santiago del Río, J. Ramos, J. L. García-Dorado, I. Gonzalez, F. Gomez-Arribas, and J. Aracil, "Packet storage at multi-gigabit rates using off-the-shelf systems," in *Proceedings of IEEE Conference on High Performance and Communications*, 2014, pp. 486–489.

[14] L. Deri, A. Cardigliano, and F. Fusco, "10 Gbit line rate packet-to-disk using n2disk," in *Proceedings of Traffic Monitoring and Analysis Workshop*, 2013, pp. 441–446.

[15] V. Uceda, M. Rodriguez, J. Ramos, J. L. García-Dorado, and J. Aracil, "Selective capping of packet payloads for network analysis and management," in *Proceedings of Traffic Monitoring and Analysis Workshop*, 2015, pp. 3–16.