Batch to the Future: Analyzing Timestamp Accuracy of High-Performance Packet I/O Engines

Victor Moreno, Pedro M. Santiago del Río, Javier Ramos, Jaime J. Garnica, and José Luis García-Dorado, *Member, IEEE*

***Abstract*—Novel packet I/O engines allow capturing traffic at multi-10Gb/s using only-software and commodity-hardware systems. This is achieved thanks to the application of techniques such as batch processing. Nevertheless, this feature involves degradation in the timestamp accuracy, which may be relevant for monitoring purposes. We propose two different approaches to mitigate such effect: a simple algorithm to distribute inter-batch time among the packets composing a batch, and a driver modification to poll NIC buffers avoiding batch processing. Experimental results, using both synthetic and real traffic, show that our proposals allow capturing accurately timestamped traffic for monitoring purposes at multi-10Gb/s rates.**

***Index Terms*—High-performance packet capture engines; packet timestamping; commodity hardware; batch processing.**

I. Introduction

THE last years have witnessed an undoubtedly explosion of the Internet users' demands for bandwidth. To manage such new demand and, especially, provide the adequate quality of service, ISPs have understood the importance of accurately monitoring their traffic, investing a great deal of effort in terms of funds and time. Few years ago, traffic monitoring at rates ranging from 100 Mb/s to 1 Gb/s was considered a challenge, whereas contemporary commercial routers feature 10 Gb/s interfaces, reaching aggregated rates as high as 100 Tb/s [1]. As a consequence, specialized hardware-based solutions such as NetFPGA or Endace DAG cards, as well as other ad-hoc solutions, have been used to such a challenging task.

Alternatively, in recent years, the research community has started to explore the use of commodity hardware together with only-software solutions as a more flexible and economical choice. This interest has been strengthened by multiple examples of real-world successful implementations of high performance capturing systems over commodity hardware [2], [3], [4]. Such approaches have shown that the keys to achieve high performance are the efficient memory management and low-level hardware interaction. However, modern operating systems are not designed with this in mind but optimized for general purpose tasks such as Web browsing or hosting. Studies about Linux network stack performance, as [5], have shown that the major flaws of standard network stack consists in per-packet resource (de)allocation and multiple data copies across the network stack. At the same time, modern NICs implement novel hardware architectures such as RSS (Receive Side Scale). RSS provides a mechanism for dispatching incoming packets to different receive queues which allows the parallelization of the capture process. In this light, novel capture engines take advantage of the parallelism capacities of modern NICs and have been designed to overcome the above mentioned deficiencies. That is, these capture engines have tuned the standard network stack and drivers to implement three improvements: (i) per-packet memory pre-allocation, (ii) packet-level batch processing and (iii) zero-copy accesses between kernel and user space.

While these improvements boost up the performance of packet capture engines, surprisingly, packet timestamp capabilities have been shifted to the background, despite their importance in monitoring tasks. Typically, passive network monitoring requires not only capturing packets but also labeling them with their arrival timestamps. In fact, the packet timestamp accuracy is relevant to the majority of monitoring applications but it is essential in those services that follow a temporal pattern. As an example, in a VoIP monitoring system, signaling must be tracked prior to the voice stream. Moreover, the use of packet batches as a mechanism to capture traffic causes the addition of a source of inaccuracy in the process of packet timestamping. Essentially, when a high-level layer asks for packets the driver stores and forwards them to the requestor at once. Therefore, all the packets of a given batch have nearby timestamps whereas inter-batch times are huge, not representing real interarrival times. This phenomenon has not received attention to date.

Consequently, this paper assesses the timestamp accuracy of novel packet capture engines and proposes two different approaches to mitigate the impact of batch processing. Specifically, (i) two simple algorithms to distribute inter-batch time among the packets composing a batch (UDTS/WDTS), and (ii) a driver modification using a kernel-level thread which constantly polls NIC packet buffers and avoids batch processing (KPT). Finally, our results, using both synthetic traffic and real traces, highlight the significant timestamping inaccuracy added by novel packet I/O engines, and show how our proposals overcome such limitation. These proposals allow us to capture correctly timestamped traffic for monitoring purposes at multi-10Gb/s rates by means of several 10 Gb/s NICs or multi-queue processing for faster interfaces such as 40 Gb/s.

II. Problem Statement

Dealing with high-speed networks claims for advanced timing mechanisms. For instance, at 10

Manuscript received June 29, 2012. The associate editor coordinating the review of this letter and approving it for publication was H. Luo.

The authors are with the High Performance Computing and Networking Research Group, Universidad Autónoma de Madrid, Spain (e-mail: {victor.moreno, pedro.santiago, javier.ramos, jaime.garnica, jl.garcia}@uam.es).

Digital Object Identifier 10.1109/LCOMM.2012.12.121433

Gb/s a 60-byte sized packet is transferred in 67.2 ns: $(60 + 4\ \text{(CRC)} + 8\ \text{(Preamble)} + 12\ \text{(Inter-Frame Gap)}) \cdot 8 \cdot 10^{-10}$, whereas a 1514-byte packet in 1230.4 ns. In the light of such demanding figures, packet capture engines should implement timestamping policies as accurate as possible.

All capture engines suffer from timestamp inaccuracy due to kernel scheduling policy because other higher priority processes make use of CPU resources. Such problem becomes more dramatic when batch timestamping is applied. In that case, although incoming packets are copied into kernel memory and timestamped in a 1-by-1 fashion, this copy-and-timestamp process is scheduled in time quanta whose length is proportional to the batch size. Thus, packets received within the same batch will have an equal or very similar timestamp. In Fig. 1 this effect is exposed for a 100%-loaded 10 Gb/s link in which 60-byte packets are being received using PacketShader [2], i.e., a new packet arrives every 67.2 ns (black dashed line). As shown, packets received within the same batch do have very little interarrival time (corresponding to the copy-and-timestamp duration), whereas there is a huge interarrival time between packets from different batches. Therefore, the measured interarrival times are far from the real values.

Figure 2 shows the standard deviation of the observed timestamp error after receiving 1514-byte sized packets for one second at maximum rate. The timestamp accuracy is degraded with batch size. Note that when the batch size is beyond 16 packets, the error tends to stall because the effective batch size remains almost constant —although a given batch size is requested to the driver, user applications will be only provided with the minimum between the batch size and the number of available packets. We notice that PFQ [4] does not use batch processing at driver-level and this source of inaccuracy does not affect its timestamping. However, timestamp inaccuracy may be added due to the per-packet processing latency.

At the same time, other sources of inaccuracy appear when using more than one hardware queue and trying to correlate the traffic dispatched by different queues. On the one hand, interarrival times may even be negative due to packet reordering as shown in [6]. On the other hand, the lack of low-level synchronism among different queues must be taken into account as different cores of the same machine cannot concurrently read the timestamp counter register [7]. PFQ suffers from these effects because it must use multiple queues in order to achieve line-rate packet capture. However, batch-oriented drivers, such as PacketShader, are able to capture wire-speed traffic using just one hardware queue.

Although Linux can timestamp packets with sub-microsecond precision by means of kernel `getnstimeofday` function, drift correction mechanisms must be used in order to guarantee long-term synchronization. This is out of the scope of this paper as it has already been solved by methods like NTP (Network Time Protocol), LinuxPPS or PTP (Precision Time Protocol) [8].

III. Proposed Solutions

To overcome the problem of batch timestamping, we propose three approaches. The first two ones are based on distributing the inter-batch time among the different packets composing a batch. The third approach adopts a packet-oriented paradigm in order to remove batch processing without degrading the capture performance.

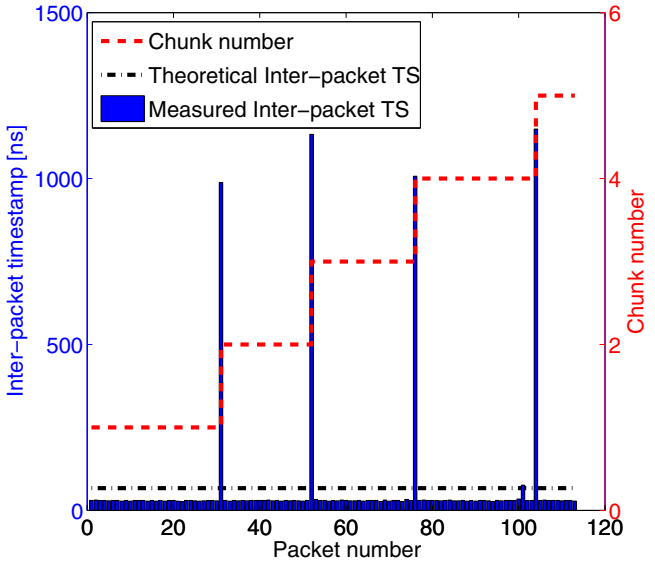


Fig. 1: Batch timestamping.

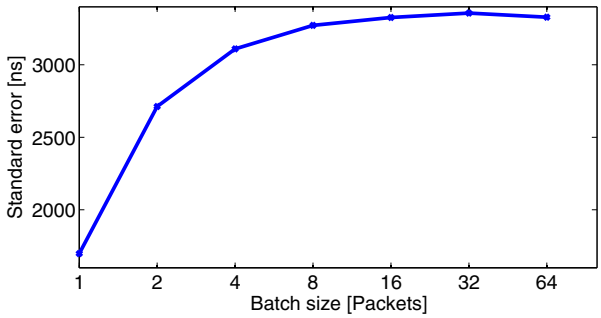


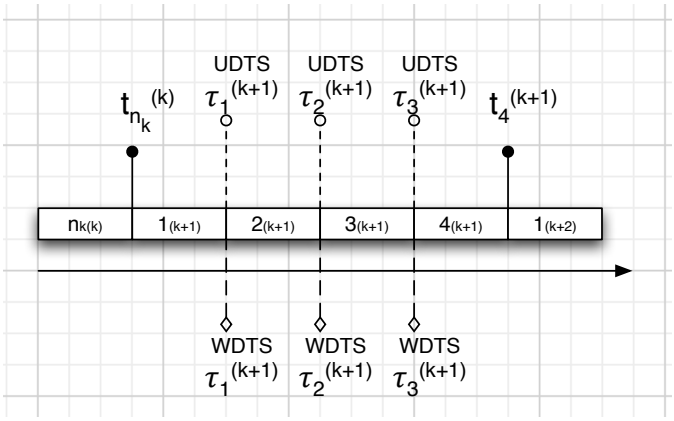
Fig. 2: Accuracy timestamp degradation with batch size.

A. UDTS: Uniform Distribution of TimeStamp

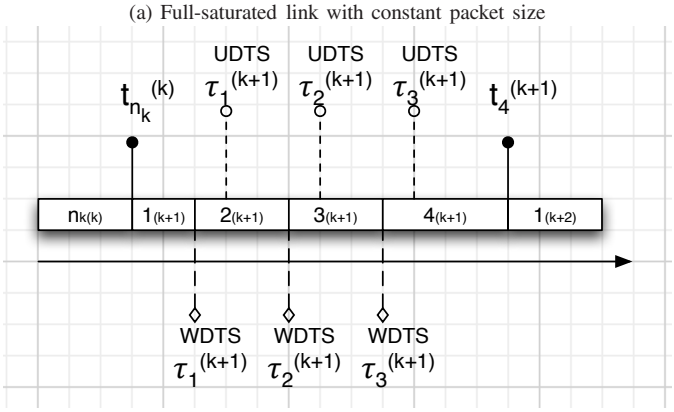
The simplest technique to reduce the huge time gap between batches is to uniformly distribute inter-batch time among the packets of a batch. Equation 1 shows the timestamp estimation of the i-th packet in the $(k+1)$-th batch, where $t_m^{(j)}$ is the timestamp of the m-th packet in the j-th batch and n_j is the number of packets in batch j.

$$\tau_i^{(k+1)} = t_{n_k}^{(k)} + \left(t_{n_{k+1}}^{(k+1)} - t_{n_k}^{(k)}\right) \cdot \frac{i}{n_{k+1}} \qquad (1)$$
$$i \in \{1, \ldots, n_{k+1}\}$$

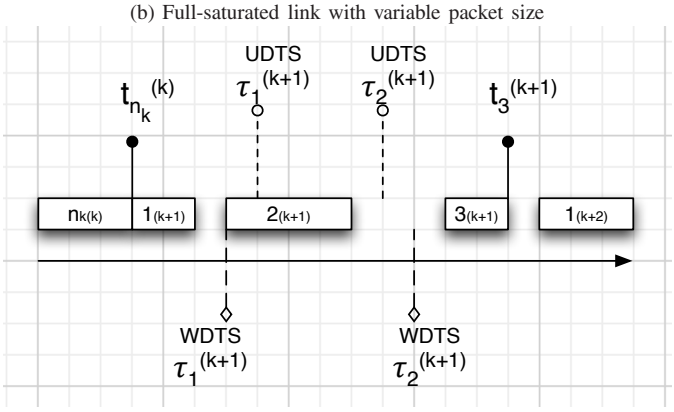
As shown in Fig. 3a, this algorithm performs correctly when the incoming packets of a given batch have the same size. A drawback of this solution is that all packets of a given batch have the same inter-arrival time regardless of their size (see Fig. 3b). Note that the inter-packet gap is proportional to the packet size when transmitting packets at maximum rate.



(a) Full-saturated link with constant packet size



(b) Full-saturated link with variable packet size



(c) Not full-saturated link with variable packet size

Fig. 3: Inter-packet gap distribution.

B. WDTS: Weighted Distribution of TimeStamp

To overcome the disadvantage of previous solution, we propose to distribute time among packets proportionally to the packet size. Equation 2 shows the timestamp estimation using this approach, where $s_j^{(k+1)}$ is the size of the j-th packet in the $(k+1)$-th batch.

$$\tau_i^{(k+1)} = t_{n_k}^{(k)} + \left(t_{n_{k+1}}^{(k+1)} - t_{n_k}^{(k)}\right) \cdot \frac{\sum_{j=1}^{i} s_j^{(k+1)}}{\sum_{j=1}^{n_{k+1}} s_j^{(k+1)}} \qquad (2)$$
$$i \in \{1, \ldots, n_{k+1}\}$$

WDTS is especially accurate when the link is completely loaded because there are no inter-packet gaps (excluding transmission time), regardless the packet size is variable, as shown in Fig. 3b. However, when the link load is lower, both UDTS and WDTS present poorer results as they distribute real inter-packet gaps among all the packets in the batch (see Fig. 3c). That is, the lower the inter-packet gap is, the higher the accuracy is.

C. KPT: Kernel-level Polling Thread

Towards a timestamping approach that performs properly regardless the link load, we propose a redesign of the network driver architecture. Novel packet capture engines fetch packets from the NIC rings only when a high-level layer polls for packets, then they build a new batch of packets and forward it to the requestor. This architecture does not guarantee when will the fetcher thread be scheduled and consequently, a source of uncertainty is added to the timestamping mechanism.

Our proposal is to create a kernel thread per each NIC's receive queue that will be constantly polling their corresponding receive descriptor rings, reading the first available descriptor flags to check whether it has already been copied into host memory. If the poll thread detects that there is one or more available packets at the receive ring, they will be copied in a 1-by-1 basis to the poll thread's corresponding circular buffer. Just before each packet copy is made, the poll thread will probe for the system time by means of the Linux kernel `getnstimeofday()` function.

A high-level application will request the packets stored in the kernel buffer by means of `read` calls over a character device file, but the timestamping process will no longer be dependent on when applications poll for new packets. This approach reduces the scheduling uncertainty as the thread will only leave execution when there are no new incoming packets or a higher priority kernel task needs to be executed. KPT causes a higher CPU load due to its busy waiting approach, but it does not degrade the performance to the point that packets are lost.

IV. Accuracy evaluation

Our setup consists of two servers (one for traffic generation and the other for receiving traffic) directly connected through a 10 Gb/s fiber-based link. The receiver has two six-core Intel Xeon E52630 processors running at 2.30 GHz with 124 GB of DDR3 RAM. The server is equipped with a 10GbE Intel NIC based on 82599 chip, which is configured with a single RSS queue to avoid multi-queue side-effects, such as reordering or parallel timestamping. The sender uses a HitechGlobal HTG-V5TXT-PCIe card which contains a Xilinx Virtex-5 FPGA (XC5VTX240) and four 10GbE SFP+ ports. Using a hardware-based sender guarantees accurate timestamping in the source. For traffic generation, two custom designs have been loaded allowing: (i) the generation of tunable-size Ethernet packets at a given rate, and, (ii) the replay of PCAP traces at variable rates.

As first experiment, we assess the timestamp accuracy sending traffic at maximum constant rate. Particularly, we send 1514-byte sized packets at 10 Gb/s, i.e., 812,744 packets per second and measure the interarrival times in the receiver side. Table I shows the error of the measured timestamp (i.e., the difference between the original and the observed interarrival times), in terms of mean and standard deviation, for a 1-second experiment for the different reviewed methods. Note

TABLE I: Experimental timestamp error (mean and standard deviation). Synthetic traffic: 1514-bytes packets

Solution	Batch size	$\bar{\mu} \pm \bar{\sigma}$ [ns]
User-level batch TS	1	1.77 ± 1765.37
	32	1.76 ± 3719.82
Driver-level batch TS	1	1.77 ± 1742.09
	32	1.77 ± 3400.72
PFQ	-	1.68 ± 13558.65
UDTS	32	1.78 ± 167.00
WDTS	32	1.77 ± 170.95
KPT	-	1.77 ± 612.72

TABLE II: Experimental timestamp error (mean and standard deviation). Real traffic: Wire-speed and Original speed

Solution	Wire-Speed $\bar{\mu} \pm \bar{\sigma}$ [ns]	Original Speed $\bar{\mu} \pm \bar{\sigma}$ [ns]
Driver-level batch TS	13.00 ± 3171.46	-25.95 ± 19399.08
UDTS	11.88 ± 608.75	-39.83 ± 13671.08
WDTS	5.31 ± 111.22	-41.77 ± 14893.97
KPT	-1.43 ± 418.42	-43.44 ± 1093.16

that the lower the standard deviation is, the more accurate the timestamping technique is. The first two rows show the results for PacketShader, chosen as a representative of batch-based capture engines. We tested with different batch sizes and different timestamping points: at user-level or at driver-level. PFQ results are shown in the following row whereas the three last ones show the results of our proposed solutions. It can be observed that timestamping error grows with batch size, as shown in Fig. 2. However, even in the best case (using one-packet batches), the error is greater than the one observed using our proposals. UDTS and WDTS methods enhance the accuracy, decreasing the standard deviation of the timestamp error below 200 ns. Both methods present similar results because all packets have the same size in this experiment. KPT technique reduces the standard deviation of the error up to $\sim$600 ns. Despite timestamping packet-by-packet, PFQ shows a timestamp standard error greater than 13 μs.

In the next experiments, we evaluate the different techniques using real traffic from a Tier-1 link (i.e., a CAIDA OC192 trace [9]). We perform two experiments: in the first one, the trace is replayed at wire speed (that is, at 10 Gb/s), and then, we replay the trace at the original speed (i.e., at 564 Mb/s, respecting inter-packet gaps). Due to storage limitation in the FPGA sender, we are able to send only the first 5,500 packets of the trace. Table II shows the comparison of the results for our proposals and the driver-level batch timestamping. We have used a batch size of 32 packets because 1-packet batches do not allow achieving line-rate performance for all packet sizes. In wire-speed experiments, WDTS obtains better results than UDTS due to different sized packets in a given batch. When packets are sent at original speed, WDTS is worse than KPT because WDTS distributes inter-packet gap among all packets. This effect does not appear at wire-speed because there is no inter-packet gap (excluding transmission time). In any case, driver-level batch timestamping presents the worst results, even in one order of magnitude.

V. Conclusion

Batch processing enhances the capture performance of I/O engines at the expense of packet timestamping accuracy. We have proposed two approaches to mitigate timestamping degradation: (i) UDTS/WDTS algorithms that distribute the inter-batch time gap among the different packets composing a batch and (ii) a redesign of the network driver, KPT, to implement a kernel-level thread which constantly polls the NIC buffers for incoming packets and then timestamps and copies them into a kernel buffer one-by-one. In fact, we propose to combine both solutions according to the link load, i.e., using WDTS when the link is near to be saturated distributing timestamp in groups of packets and, otherwise, using KPT timestamping packet-by-packet. We have stress tested the proposed techniques, using both synthetic and real traffic, and compared them with other alternatives achieving the best results (standard error of 1 μs or below).

To summarize, we alert research community to timestamping inaccuracy introduced by novel high-performance packet I/O engines, and proposed two techniques to overcome or mitigate such issue.

Acknowledgements

We would like to thank Prof. Javier Aracil and Prof. Francisco J. Gomez-Arribas for their valuable comments and recommendations.

References

[1] Cisco, "Cisco carrier routing system," http://cisco.com/en/US/products/ps5763/index.html.

[2] S. Han, K. Jang, K. S. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," *ACM SIGCOMM Computer Commun. Rev.*, vol. 40, no. 4, pp. 195–206, 2010.

[3] L. Rizzo, M. Carbone, and G. Catalli, "Transparent acceleration of software packet forwarding using netmap," in *Proc. 2012 IEEE INFOCOM*.

[4] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On multi-gigabit packet capturing with multi-core commodity hardware," in *Proc. 2012 Passive and Active Measurement Conference*.

[5] G. Liao, X. Znu, and L. Bnuyan, "A new server I/O architecture for high speed networks," in *Proc. 2011 IEEE Symposium on High-Performance Computer Architecture*.

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

[7] T. Broomhead, J. Ridoux, and D. Veitch, "Counter availability and characteristics for feed-forward based synchronization," in *Proc. 2009 IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*.

[8] M. Laner, S. Caban, P. Svoboda, and M. Rupp, "Time synchronization performance of desktop computers," in *Proc. 2011 IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*.

[9] C. Walsworth, E. Aben, K. Claffy, and D. Andersen, "The CAIDA anonymized 2009 Internet traces," http://www.caida.org/data/passive/passive_2009_dataset.xml.