2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)

Artificial Accurate Time-stamp in Network Adapters

Wojciech Waśko
Nvidia
Młynek, Poland
wwasko@nvidia.com

Dotan David Levi
Nvidia
Yokne'am, Israel
dotanl@nvidia.com

Teferet Geula
Nvidia
Jerusalem, Israel
tgeula@nvidia.com

Amit Mandelbaum
Nvidia
Jerusalem, Israel
amitma@nvidia.com

***Abstract*—Nanosecond-level synchronization of network nodes is an enabler for a wide range of distributed applications. The achievable accuracy of network synchronization is bounded by measurement errors in both packet ingress and egress time-stamps provided by the network adapters. Today's commodity network adapters provide time-stamps with an accuracy of, at best, 100s of nanoseconds. Improving the accuracy rate of the time-stamp would require a new hardware design and take years to deploy. In this paper, we present Artificial Accurate Time-stamp (AAT), a machine-learning-based software method to improve the accuracy of hardware time-stamps beyond what is provided by the underlying hardware design. We successfully apply this method to remove time-stamping errors caused by varying network traffic patterns and queue variance in the network adapter. Additionally, we demonstrate a real-world application that uses this method to reduce network synchronization error, from over 300 nanoseconds to lower than 30 nanoseconds (a 10x reduction), without changing the hardware or any software in the application layer.**

***Index Terms*—networking, network adapters, Machine learning, AI, time-stamping, synchronization, clock-synchronization PTP**

I. Introduction

Clock synchronization has been a key long-standing challenge in packet-based distributed systems. IEEE 1588 [1] has evolved dramatically in recent years to accommodate an increasing number of use cases. Some examples: In the domain of 5G telecommunications infrastructures, where timing synchronization requirements can be very strict [2]–[4]; another example is autonomous cars where a dedicated PTP profile [5] has been developed exclusively for timing synchronization [6], [7]. In data centers, however, although the topic of timing synchronization had been raised more than a decade ago [8], [9], until recently, there was no publicly announced deployment of PTP in data centers. Not long ago, the Open Compute Project (OCP) established a project (code-named TAP) [10], with the vision and target to define the time-synchronization server for modern high-scale data centers. The Project defines the requirements in order to deploy time synchronization service, and explore use cases to exploit/leverage it. For example, one of the use cases leveraging the time service in a data center is distributed database [11] that use clock synchronization to enforce external consistency. It is commonly known that the accuracy of the clock synchronization has been bound by hardware time-stamping accuracy [12]. Scientific solutions such as White Rabbit [13] would require a complete redesign of the infrastructure as they rely on super-precise custom hardware implementations, which dictate using either dedicated hardware for time-sensitive networking or an entirely new architecture. In data centers, however, the main challenge has been the huge install-base of network interface controllers (NICs) and switches; most of which lack accurate PHY layer time-stamping. Replacing such an extensive install base would pose a significant financial burden, which seems to be the biggest hurdle to deploying an accurate time service. There have been several attempts to overachieve the accuracy of hardware time-stamping, such as the Huygens framework [14], [15]. This and other suggested frameworks [16], [17] dramatically abandon PTP (and NTP) protocols, modify the idea of a clock tree, and thus require a substantial re-coding of application and infrastructure software layers.

In this work, we address those challenges and demonstrate machine learning methods to improve the accuracy of hardware time-stamping. To the best of our knowledge, this is the first work to discuss and propose a method for improving hardware time-stamping accuracy without modifying the hardware, driver, and application layers. This work can be implemented on an existing PTP deployment, or even Network Time Protocol (NTP) [18]. Specifically, we make the following key contributions: 1) We expose and elaborate the fact that the inaccuracy of a hardware time-stamp is dependent on the internal hardware state; 2) we explain how to train an AI system to correlate between the hardware state and the inaccuracy; and 3) we apply our technique to create a solution which dramatically improves hardware time-stamping without any architectural changes. The result is significant and demonstrates a magnitude of order improvement (from 300 to sub-30 nanoseconds). The results were tested using ptp4l [19], [20], over real-world traffic and synthetic traffic (using a traffic generator to emulate some extreme cases). Lastly, our work enables using existing network adapters for the most challenging PTP profiles (G.8273.2 class C) [21], enabling the use of workloads and architecture, which was previously not feasible, (e.g., softwarization of the 5G infrastructure) due to strict real-time restrictions.

A. AI in Networking

The role of Machine Learning (ML) in networking has grown significantly in recent years (see surveys [22], [23]). The works in this field employ the full range of ML techniques, including supervised learning, unsupervised learning, and reinforcement learning (RL), address a variety of net-

978-0-7381-2646-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00218

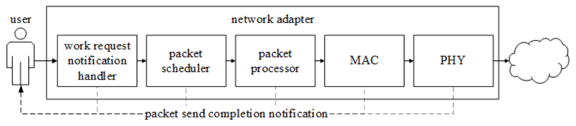

user
network adapter
work request notification handler
packet scheduler
packet processor
MAC
PHY
packet send completion notification

Fig. 1. Example of a NIC transmit pipeline

working problems. In the area of supervised learning we can find, for example, works on traffic prediction [24]–[26], load prediction [27], [28] and classification of network traffic [29], [30], or even specific packets [31], [32]. Unsupervised learning for example, is used in the field of cyber security [33]–[35], or for detecting anomalous performance behaviors [36], [37]. Finally, examples of using reinforcement learning techniques can be found in the fields of congestion control, [38] adaptive routing [39]–[41] and cyber security [42]–[45].

While these works have shown that ML methods can help to achieve improved performance in the area of networking, most of them suffer from two main issues: The first is computational overhead, including various resource management problems in practical deployment [22]. The second issue is that the majority of the ML-based solutions are evaluated solely on the basis of the synthetic data obtained via simulations [46] and lab experiments, which may differ significantly from real-world data [47]. Our approach overcomes both issues: Our ML model requires very little computational resources, while still achieving the desired accuracy, and can be implemented on real hardware, without an overhead. Most importantly, it is evaluated on a diverse set of real-world traffic patterns.

II. Network Adapter Pipeline and Time-stamping

High-speed commodity network adapters are complex machines capable of sending and receiving hundreds of millions of packets per second simultaneously, with link speeds in excess of 100 Gbps, all while applying sophisticated filtering, steering and packet processing. The ever-growing performance requirements have driven a design that decouples the packet processing pipeline into multiple stages.

The NIC pipeline is a system composed of multiple interconnected queues, most of which have fixed-size input buffers. Some of these queues also contain multiple internal units performing work in parallel. Packets being processed constitute jobs in the system; typically they are assigned priorities such that at each individual stage, the hardware can pick the job (packet) with the highest priority. Additionally, queues in this system can destroy jobs (e.g., a dropped packet due to firewall rules) or create new jobs (e.g., replicating packets in case of multicast communication).

In the following section, using the example of packet transmission, we provide a high-level overview of the different stages involved in a network adapter's operation, and how they relate to packet time-stamping. We have chosen the transmit pipeline as it presents the most difficult challenge in the field of packet time-stamping.

A. NIC TX Pipeline Overview

A user application submits (through a library or a device driver) a packet send request. This request is almost immediately acknowledged by the network adapter front end. Very little processing is performed at this stage, besides that of basic security and permission checks. From there, the packet send request enters a packet scheduler, which determines when that packet should be sent. This is typically where Quality of Service (QoS) functionality is implemented. Once the scheduler determines that it is the user packet's turn to be sent, the packet moves into the processing stage where various offloads are performed. Examples of such offloads include packet encryption (IPSec [48] or TLS [49]), complex steering [50], tunnel encapsulation [51], and header rewrite. The following stage in the pipeline is the Medium Access Control (MAC), which is responsible for both managing and scheduling packet transmissions on the physical medium and is also typically responsible for calculating and appending the packet's frame check sequence. The last stage in the pipeline is the physical layer that converts the packet bits to electrical symbols, which are finally transmitted on the physical medium.

B. Impact of Network Adapter Design on the Accuracy of Transmit Time-stamps

An important issue that needs to be addressed in each network adapter's design is when to notify the user that their packet has been sent. In principle, any of the stages in the transmission pipeline may be used to generate a send completion notification, along with its associated packet transmission time-stamp. The packet send notification must be delivered to the entity responsible for sending the packet. However, given the high speed of the pipeline that is handling millions of packets per second, and the fact that the entity requesting the transmit time-stamp is just one of thousands of NIC users, with each potentially residing in a different thread, process, application, CPU core, or even virtual machine [52], [53], the implementation can quickly get complicated. The earlier in the pipeline the packet send notification is returned, the simpler the network adapter's pipeline becomes; it requires fewer resources and less buffering. However, the resulting transmit time-stamp is less accurate. Conversely, though generating the send completion notification at the physical layer provides the highest possible accuracy of time-stamps, it is also the most costly approach since such design needs to maintain the association between packets and the requesting entities throughout the entire network adapter. This design choice represents a decision based on the feature's return on investment (ROI); high-speed commodity network adapters available on the market typically attempt to strike a balance by considering the packet to have been sent (and generating the respective notification containing the transmit time-stamp) in either the packet scheduler or the packet processor.

C. *Sources of Inaccuracies in Transmit Time-stamps*

Any variable delay that a packet encounters between the moment it is time-stamped and when it actually egresses on the physical medium adds to the inaccuracy of the transmit time-stamp. A list of potential reasons why a packet might be delayed includes:

- **Congestion Control** [54], [55]: congestion control protocols can intentionally delay transmission of a packet in order to reduce overall network congestion
- **Packet Processing** [53], [56]: e.g. packet encryption method in which execution time depends on the packet size
- **Traffic burstiness and the resulting queuing and buffering** [56]–[58]: network traffic patterns have been found to be fundamentally bursty at small timescales, leading to buffering and queuing effects in the network hardware
- **Hardware resources arbitration and packet QoS** [59], [60]: other packet(s) with higher priority are processed and transmitted before the time-stamped packet or the time-stamped packet awaits hardware resources currently occupied by other packets

III. Improving time-stamp Accuracy

A. *Solution Overview*

As explained, the delay encountered by a specific packet depends on the NIC state as the packet is being processed. Hardware implementations typically have probes in the form of performance counters or queue depth indications used by hardware designers to inspect the performance of the blocks they create. We re-purpose those hardware probes in order to introduce observability of the NIC's instantaneous state while the packets are processed by NIC's pipeline. Further, we employ Machine Learning (ML) algorithms to estimate the delay encountered by a specific packet from the moment it's time-stamped to the moment it egresses on the wire, based on the values of those hardware probes. The time-stamp provided by hardware is then adjusted by this estimated delay, resulting in an **Artificial Accurate Time-stamp (AAT)**.

Fig. 2 illustrates the overall architecture of the solution. We created a firmware daemon that continuously queries the state of the NIC transmit pipeline and reports it to a time series database. The hardware probes its queries, including queue occupancy and buffer utilization, as well as instantaneous packet and bit rate. Because the daemon runs on the network adapter itself, it has a very low latency for querying the hardware state. The data in the time series database is then used by an interposer library which intercepts an application's (e.g. ptp4l [19]) request for a packet transmit time-stamp. This library obtains the raw, hardware-generated transmit time-stamp and looks up the hardware state snapshots that were taken immediately before and after the generation of the packet transmit time-stamp. Using those hardware state snapshots, the interposer library executes a machine learning model to estimate the delay that the packet experiences before its egress

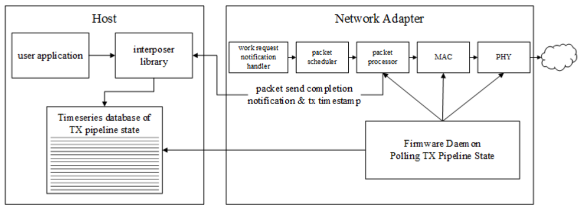

Host
user application
interposer library
Timeseries database of TX pipeline state
Network Adapter
work request notification handler
packet scheduler
packet processor
MAC
PHY
packet send completion notification & tx timestamp
Firmware Daemon Polling TX Pipeline State

Fig. 2. Architecture of solution

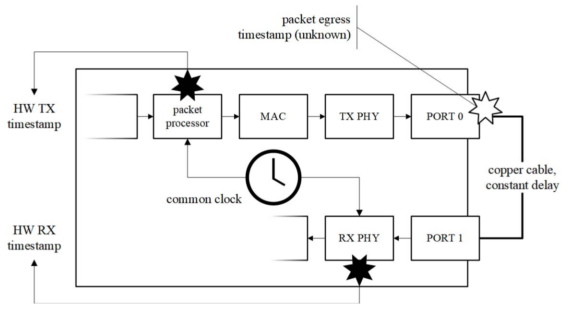


Fig. 3. External loopback setup for obtaining training data

on the wire. It then adds the estimated delay to the hardware-generated transmit time-stamp and finally returns the modified time-stamp to the application. Thanks to this architecture, no application changes are required in order to use the improved time-stamps.

B. *Obtaining the Training Data*

In order to build the machine learning model, we need to find the ground truth for when the packets are transmitted on the wire and contrast that with the hardware-generated transmit time-stamp. For that purpose, we built the system shown in figure 3. The system is composed of a single NVIDIA® Mellanox® ConnectX®-6 Dx [61], a two-port network adapter in which the ports are connected with an external copper cable so that any traffic transmitted from port 0 is received on port 1. In the case of the adapter used in our experiments, the receive time-stamp accuracy does not depend on the traffic pattern as the receive time-stamping is performed in the physical layer. Importantly, both ports use the same underlying clock source so that both hardware transmit time-stamps are expressed on the same time-scale. Since copper cables exhibit a known constant delay that can be compensated, the receive time-stamp on port 1 becomes a proxy measurement of the packet's egress from port 0.

We have created a special application that sends packets on port 0, obtains their transmit time-stamps (as reported by hardware), looks up the hardware state around the transmission time (hardware state snapshots taken immediately before and after the packet's transmit time-stamp), and also receives the same packet on port 1 along with its receive time-stamp. All this data is then saved into CSV files for further processing.

In order to simulate the challenging scenarios this solution would be used in, we have created a traffic generator application based on DPDK [62] that produces various traffic patterns. The following packet size distributions were implemented in this application:

- mix: 29x 90B, 1x 92B, 12x 594B, 8x 1418B
- uniform: 1x 64B, 1x 65B, ..., 1x 1518B
- fixed: all packets of the same (configurable) size

Traffic pattern drawing from any of these packet size distributions can be generated with instantaneous bandwidth (measured over a time period of 260 milliseconds) freely fluctuating between 0 Gbps and full link speed.

We then carefully examined the details of NVIDIA® Mellanox® ConnectX®-6 Dx [61] design, identifying all blocks which may delay packet's egress after it has already been time-stamped. This list includes one output queue in the packet processor, two queues in the MAC layer and two additional queues in the PHY layer. We have then selected hardware probes that provide observability into the state of those blocks; the values of these probes around the packet's raw hardware TX time-stamp were included among the features for training the machine learning models (see Table I).

IV. Model Training and Selection

Dataset: Using the mechanism described above, we generated a dataset composed of 12.7M observations for different NIC states, explicitly different link speeds and interfering traffic patterns. The dataset was divided into train (70%) and test (30%) datasets. Each observation included the NIC's state counter values for time samples before and after the observation packet processing. In addition, we used the time that has elapsed between sampling the counters and the packet being processed, which added two more values to the observation. A full feature list can be found in Table I.

Fig. 4 depicts the distribution of the time delay (the target) for different link speeds. We can see that higher link speeds result in shorter time delays. The behaviour of the data suggests that a tree-based model is a natural fit, in which a smaller sub-tree is trained for each hardware state.

Training: We compared the results of 4 ML regression algorithms: Linear Regression, Decision Tree, CatBoost, and Neural Network, trained with the L2 regression loss. We evaluated and compared the results by calculating the Mean Absolute Error (MAE) on a test set:

- Linear regression: A simple linear model. We used this model as a baseline, to justify more complicated ML models.
- Decision Tree: A single and deep decision tree with depth $d_{DT} = 20$.
- CatBoost [63]: This ML model is an assembly of small "weak" trees. We trained the model with $number\_of\_trees = 2844$ trees, each of depth $d_C = 6$.
- Neural Network: We trained a NN model to test whether it can outperform a tree based model. First, the data is uniformly normalized between [-1,1]. We used a deep neural network (DNN) with 4 hidden layers, Adam optimizer, $learning\_rate = 0.002$, $weight\_decay = 5e-5$, $batch\_size = 1024$ and $epochs = 500$. We utilized the PyTorch framework for training [64].

TABLE I
List of features in each of "before" and "after" time-samples

Feature	Description
sampling_delta	Time difference between taking the hardware state sample and packet TX time-stamp
pkt_processor_ outstanding_bytes	Number of bytes already processed and awaiting final arbitration and submission to TX MAC
MAC_in_bytes	Number of bytes buffered at TX MAC input
MAC_in_packets	Number of packets buffered at TX MAC input
MAC_xcode_bytes	Number of bytes buffered in an internal TX MAC transcoding queue
PHY_in_bytes	Number of bytes buffered at TX PHY input
PHY_internal_bytes	Number of bytes in additional stage of buffering inside TX PHY

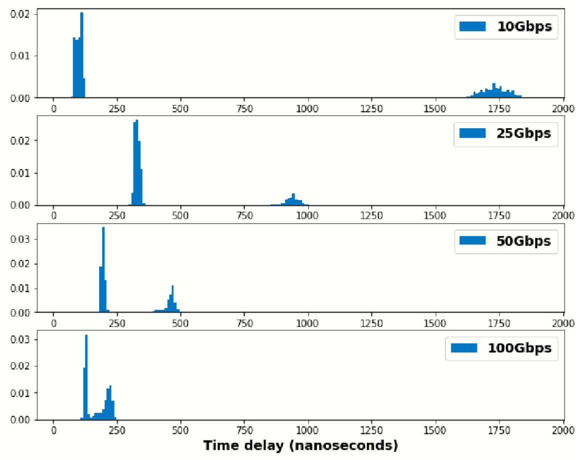


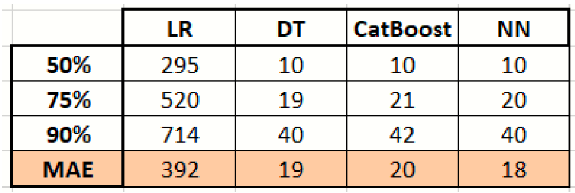
Fig. 4. Target distribution for different link speeds

Testing Results and Model Selection: Below we can see the MAE results on the test dataset for the different ML algorithms:

- Linear Regression: 392ns
- Decision Tree: 19ns
- CatBoost: 20ns
- Neural Network: 18ns

Full results table can be seen in figure 5.

It is clear that a simple linear regression model is not complex enough to capture the correlation/relationship between the NIC state and the time delta, and a more sophisticated ML model is required. The other ML algorithms, on the other hand, managed to reduce the error from 392ns to ~20ns MAE. While there is a little difference between the NN and the tree-based models' results, the NN is more complex to train and has more hyper-parameters to optimize, whereas tree-based models have a simpler training process since they don't require data normalization and have very short inference times. If we want a model to be retrained for changes in



	LR	DT	CatBoost	NN
50%	295	10	10	10
75%	520	19	21	20
90%	714	40	42	40
MAE	392	19	20	18

Fig. 5. Absolute error percentiles (nanoseconds) evaluated on test set for Linear Regression (LR), Decision Tree (DT), CatBoost, and Neural Network (NN)

the hardware or the data, we should prefer a simpler training process such as the one required for training tree-based models. In addition, tree-based models (either CatBoost or DT) can be implemented as a chain of if/else statements on low memory budget resources. Choosing between CatBoost and DT depends on the performance requirements of the deployment system in terms of inference time and model size, for which there is a trade-off between those two characteristics. Decision Trees have a very short inference time (dominated by the depth of the tree d_{DT}), but also a large number of nodes ($\leq 2^{d_{DT}+1} - 1$, equality is achieved when the tree is full). On the other hand, a CatBoost model has a smaller model size ($\leq number\_of\_trees * (2^{d_C+1} - 1)$, depends on tree growing policy), but a longer inference time (dominated by $number\_of\_trees * d_C$). We decided to implement CatBoost as its low memory consumption provided a big advantage over DT, while its inference time was comparably the same as DT's in our case.

V. Results And Conclusions

A. *Impact on Synchronization Performance*

In order to verify the real-world applicability of our approach, we have used the interposer library in conjunction with ptp4l [19] to inspect how this solution is improving synchronization quality. For this test, we have connected two machines (host 1 and host 2) equipped with NVIDIA® Mellanox® ConnectX®-6 Dx [61] back to back, with host 1 acting as the synchronization master. We have also connected the synchronization master PPS OUT port to the PPS IN port of the other host in order to use the PPS out/in capability of NVIDIA® Mellanox® ConnectX®-6 Dx [61] (see Fig. 6). In this setup, synchronization is established over an Ethernet link. Additionally, host 1 emits an electrical signal on the PPS OUT port every time a full second elapses according to its clock. Host 2 records the time on its local clock for every electrical input it sees on its PPS IN port. By subtracting the difference between this time and the nearest rounded second, we have obtained a hardware-based measurement of time error between the two nodes.

In the test procedure, the traffic generator is started first, followed by ptp4l [20] master and slave applications launched on the respective hosts. Measurement of the time error via PPS signals begins after a 20-second delay for the initial nodes' synchronization, and lasts one minute. Finally, the maximum time error is calculated as the difference between the largest and the smallest time error.

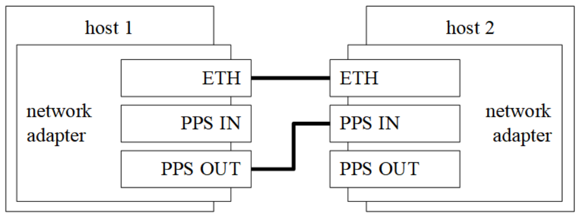


Fig. 6. Setup for measuring time error via PPS signals

TABLE II
Maximum time error with fluctuating interfering traffic. All values in nanoseconds.

link speed	traffic	no AAT	with AAT	improvement
25Gbps	none	20	24	-20%
	mix	340	28	92%
	1024B	352	20	94%
	1322B	300	20	93%
	1518B	312	24	92%
	uniform	332	24	93%
100Gbps	none	16	24	-50%
	mix	56	28	50%
	1024B	56	20	64%
	1322B	32	24	25%
	1518B	32	24	25%
	uniform	32	24	25%

Results are given in Table II. For the non-interesting case of an idle 25Gbps link, we can see that AAT has reduced the accuracy by 20% but it is still under 30ns. The major observation is that for a 25Gbps link loaded with a traffic pattern composed of mixed packet sizes, the maximum time interval error has dropped from 340 nanoseconds to 28 nanoseconds, a 92% improvement. More importantly, our solution delivers a sub-30 nanosecond time error across various link speeds and interfering traffic patterns, meeting the highly demanding G.8273.2 Class C [21] clock requirements. Results on a 100Gbps link - though still adequate - are less spectacular since the original performance under various conditions was already very good. We have also tested a packet size of 1322 bytes, indicated as a realistic use case for 5G fronthaul [65] with very good results.

For visualization purposes, we have also plotted the PPS-measured time error using a graphical tool. A visualization for a 25Gbps link is shown in Fig. 7 which is a time-series plot of the time error. The plots consist of three separate parts. The leftmost part shows the time error achieved by using raw hardware transmit time-stamps without AAT adjustments when the synchronization is performed without any interfering traffic on the link. The middle part shows the time error - still without AAT - with the traffic generator transmitting mixed traffic patterns with fluctuating bandwidth. Finally, the rightmost part shows the time error with the mixed traffic still present on the link but with the transmit time-stamps

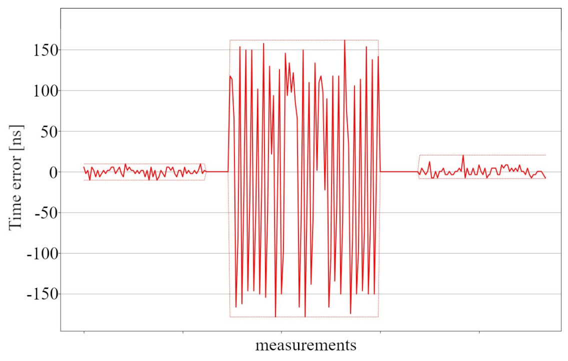


Fig. 7. Time error plot (25Gbps link). Left: idle link without AAT, middle: fluctuating mix traffic without AAT, right: fluctuating mix traffic with AAT

transparently corrected by the interposer library according to the solution described in this paper using AAT. We can clearly see that the time error reduces to below 30ns as required.

B. Inference Complexity and Resource Usage

We have profiled the runtime cost of using the AAT in order to assess its suitability for deployment in real-world solutions, including embedding it in hardware. Our experiments were run on an Intel Xeon Gold 6242 CPU running at 2.8GHz under control of RHEL 7.6. The evaluated CatBoost model contained 2844 trees and occupied 3012 kB on disk; its performance was measured over 6 thousand independent inference cycles. The minimum inference time across this measurement was 39 microseconds, the median was 73 microseconds and the maximum inference time was 260 microseconds. ITU-T 8275.1 [66], a common PTP profile used in 5G networking, dictates that time-critical messages (sync and delay requests) be exchanged every 32 milliseconds on average. Given that the longest measured inference time of our proposed method is 260 microseconds, the inference consumes in the worst case ∼0.8% of available message-to-message period which we consider to be negligible. Additionally, we have profiled memory consumption and we have found that AAT, as implemented by our inter-poser library which simply interfaces with the publicly available CatBoost library [63], has increased the memory consumption by 3871 kB. Based on these measurements, this solution could be readily deployed on advanced Smart NICs such as NVIDIA® BlueField® [67] which are equipped with capable on-board CPUs and as much as 16 GB of memory. Porting this solution to simpler, embedded devices such as NVIDIA® Mellanox® ConnectX®-6 Dx [61] is more also possible by compressing the model and implementing it as a chain of if/else statements, which can be done easily for tree-based models.

C. Applicability in Data Centers and other Domains

To conclude our paper, we expect that our work will enable softwarization and usage of existing commodity NICs in the 5G domain, as well as improve the time sync in existing data center deployments. We also believe that the method we have introduced presents a novel paradigm of using online Machine Learning inference to complement existing hardware implementations, yielding benefits far greater than what can be achieved with offline device calibration. In essence, packet transmit timestamp is nothing but a measurement of time when a given packet passes a reference plane inside the network card. We found this measurement to be tainted by noise which depends on hardware state parameters in a nonlinear way and discovered a way to eliminate this noise by means of online inference. The same philosophy can be applied regardless of whether the measurement involves time, location, force, tensile stress or any other entity. For example, a force sensor's accuracy might be simultaneously affected by how many measurement cycles it had performed before, extension of an internal spring and the oscillation of power supply voltage with all of these components contributing to the measurement error in a nontrivial way. A machine learning model might then be trained to correct this noise, ultimately resulting in a measurement which is more precise than what the underlying hardware allows. We expect that the coming years will see multiple instances of applying the same methodology to various domains.

References

[1] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Std. 1588, 2020.

[2] J.-C. Lin, "Synchronization requirements for 5g: An overview of standards and specifications for cellular networks," *IEEE Vehicular Technology Magazine*, vol. 13, no. 3, pp. 91–99, 2018.

[3] H. Li, L. Han, R. Duan, and G. M. Garner, "Analysis of the synchronization requirements of 5g and corresponding solutions," *IEEE Communications Standards Magazine*, vol. 1, no. 1, pp. 52–58, 2017.

[4] T. Jokiaho and P. Vaananen. Open synchronization implementations on linux/k8s clusters. Open Compute, TAP Project, meeting #22. [Online]. Available: https://drive.google.com/file/d/1qU87Fys2wDBNK_3kyylwOCW1J752IPzi/view?usp=sharing

[5] *IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications*, IEEE Std., June 2020.

[6] J. Fischer. (2016, January) The importance of timing to autonomous vehicle navigation. [Online]. Available: https://doi.org/10.33012/2016.13495

[7] Y. J. Kim, J. H. Kim, B. M. Cheon, Y. S. Lee, and J. W. Jeon. (2014) Performance of ieee 802.1as for automotive system using hardware timestamp.

[8] B. Liskov. (1991) Practical uses of synchronized clocks in distributed systems. New York, NY, USA. [Online]. Available: https://doi.org/10.1145/112600.112601

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2491245

[10] Time applianc project. Open Compute. [Online]. Available: https://www.opencompute.org/projects/time-appliances-project-tap-incubation

[11] G. Chalakov. Practical use cases of synchronized clocks. Open Compute, TAP Project, meeting #6. [Online]. Available: https://www.youtube.com/watch?v=Xzh3JUzbz4I

[12] P. Loschmidt, R. Exel, A. Nagy, and G. Gaderer. (2008, Sep.) Limits of synchronization accuracy using hardware support in ieee 1588.

[13] M. Lipiński, T. Włostowski, J. Serrano, and P. Alvarez. (2011, Sep.) White rabbit: a ptp application for robust sub-nanosecond synchronization.

[14] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. (2018, apr) Exploiting a natural network effect for scalable, fine-grained clock synchronization. Renton, WA. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/geng

[15] B. Prabhakar. White rabbit: An accurate time and frequency transfer over ethernet. Open Compute, TAP Project, meeting #14. [Online]. Available: https://www.youtube.com/watch?v=-uNaQENycMA

[16] E. Mallada, X. Meng, M. Hack, L. Zhang, and A. Tang, "Skewless network clock synchronization without discontinuity: Convergence and performance," *IEEE/ACM Transactions on Networking*, vol. 23, no. 5, pp. 1619–1633, 2015.

[17] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon. (2016) Globally synchronized time via datacenter networks. New York, NY, USA. [Online]. Available: https://doi.org/10.1145/2934872.2934885

[18] The network time protocol. [Online]. Available: https://www.ntp.org/

[19] The linux ptp project. [Online]. Available: http://linuxptp.sourceforge.net/

[20] The linux ptp project. linux kernel. [Online]. Available: http://linuxptp.sourceforge.net/

[21] Timing characteristics of telecom boundary clocks and telecom time slave clocks for use with full timing support from the network. ITU-T.

[22] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 16, 2018.

[23] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang, "Machine learning for networking: Workflow, advances and opportunities," *IEEE Network*, vol. 32, no. 2, pp. 92–99, 2017.

[24] Y. Sun, X. Yin, J. Jiang, V. Sekar, F. Lin, N. Wang, T. Liu, and B. Sinopoli, "Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 272–285.

[25] J. Jiang, V. Sekar, H. Milner, D. Shepherd, I. Stoica, and H. Zhang, "{CFA}: A practical prediction system for video qoe optimization," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 137–150.

[26] Y. Zhu, G. Zhang, and J. Qiu, "Network traffic prediction based on particle swarm bp neural network." *J. Networks*, vol. 8, no. 11, pp. 2685–2691, 2013.

[27] P. Bermolen and D. Rossi, "Support vector regression for link load prediction," *Computer Networks*, vol. 53, no. 2, pp. 191–201, 2009.

[28] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "Deepcog: Cognitive network management in sliced 5g networks with deep learning," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 280–288.

[29] J. Zhang, Y. Xiang, Y. Wang, W. Zhou, Y. Xiang, and Y. Guan, "Network traffic classification using correlation information," *IEEE Transactions on Parallel and Distributed systems*, vol. 24, no. 1, pp. 104–117, 2012.

[30] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu, "Robust network traffic classification," *IEEE/ACM transactions on networking*, vol. 23, no. 4, pp. 1257–1270, 2014.

[31] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 256–269.

[32] A. Rashelbach, O. Rottenstreich, and M. Silberstein, "A computational approach to packet classification," *arXiv preprint arXiv:2002.07584*, 2020.

[33] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications surveys & tutorials*, vol. 18, no. 2, pp. 1153–1176, 2015.

[34] M. Thottan and C. Ji, "Anomaly detection in ip networks," *IEEE Transactions on signal processing*, vol. 51, no. 8, pp. 2191–2204, 2003.

[35] P. Casas, J. Mazel, and P. Owezarski, "Unsupervised network intrusion detection systems: Detecting the unknown without knowledge," *Computer Communications*, vol. 35, no. 7, pp. 772–783, 2012.

[36] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–35, 2015.

[37] P. Barford, N. Duffield, A. Ron, and J. Sommers, "Network performance anomaly detection and localization," in *IEEE INFOCOM 2009*. IEEE, 2009, pp. 1377–1385.

[38] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *International Conference on Machine Learning*, 2019, pp. 3050–3059.

[39] M. Littman and J. Boyan, "A distributed reinforcement learning scheme for network routing," in *Proceedings of the international workshop on applications of neural networks to telecommunications*. Erlbaum Hillsdale, NJ, USA, 1993, pp. 45–51.

[40] S.-C. Lin, I. F. Akyildiz, P. Wang, and M. Luo, "Qos-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach," in *2016 IEEE International Conference on Services Computing (SCC)*. IEEE, 2016, pp. 25–33.

[41] G. Stampa, M. Arias, D. Sánchez-Charles, V. Muntés-Mulero, and A. Cabellos, "A deep-reinforcement learning approach for software-defined networking routing optimization," *arXiv preprint arXiv:1709.07080*, 2017.

[42] J. Cannady, "Next generation intrusion detection: Autonomous reinforcement learning of network attacks," in *Proceedings of the 23rd national information systems security conference*, 2000, pp. 1–12.

[43] R. Ganesan, S. Jajodia, A. Shah, and H. Cam, "Dynamic scheduling of cybersecurity analysts for minimizing risk using reinforcement learning," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, pp. 1–21, 2016.

[44] A. Servin and D. Kudenko, "Multi-agent reinforcement learning for intrusion detection," in *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*. Springer, 2005, pp. 211–223.

[45] T. T. Nguyen and V. J. Reddi, "Deep reinforcement learning for cyber security," *arXiv preprint arXiv:1906.05799*, 2019.

[46] T. Issariyakul and E. Hossain, "Introduction to network simulator 2 (ns2)," in *Introduction to network simulator NS2*. Springer, 2009, pp. 1–18.

[47] J. Xie, F. R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, and Y. Liu, "A survey of machine learning techniques applied to software defined networking (sdn): Research issues and challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 393–430, 2018.

[48] S. Kent and R. Atkinson, "Rfc 2401: Security architecture for the internet protocol," *Internet Engineering Task Force (IETF)*, 1998.

[49] E. Rescorla *et al.*, "Rfc 8446: The transport layer security (tls) protocol version 1.3," *Internet Engineering Task Force (IETF)*, 2018.

[50] R. C. Sofia, "A survey of advanced ethernet forwarding approaches," *IEEE Communications surveys & tutorials*, vol. 11, no. 1, pp. 92–115, 2009.

[51] I. Cerrato and F. Risso, "Enabling precise traffic filtering based on protocol encapsulation rules," *Computer Networks*, vol. 136, pp. 51–67, 2018.

[52] M. Honda, F. Huici, G. Lettieri, and L. Rizzo, "mswitch: a highly-scalable, modular software switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015, pp. 1–13.

[53] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. Lakshman, "Uno: Uniflying host and smart nic offload for flexible packet processing," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 506–519.

[54] M. Bahnasy, H. Elbiaze, and B. Boughzala, "Hetflow: A distributed delay-based congestion control for data centers to achieve ultra low queueing delay," in *2017 ieee international conference on communications (icc)*. IEEE, 2017, pp. 1–7.

[55] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 329–342.

[56] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter, "Bullet trains: A study of nic burst behavior at microsecond timescales," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, 2013, pp. 133–138.

[57] L. Angrisani, G. Ventre, L. Peluso, and A. Tedesco, "Measurement of processing and queuing delays introduced by an open-source router in a single-hop network," *IEEE transactions on instrumentation and measurement*, vol. 55, no. 4, pp. 1065–1076, 2006.

[58] H. Jiang and C. Dovrolis, "Why is the internet traffic bursty in short time scales?" in *Proceedings of the 2005 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems*, 2005, pp. 241–252.

[59] M. Pirvu, L. Bhuyan, and N. Ni, "The impact of link arbitration on switch performance," in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. IEEE, 1999, pp. 228–235.

[60] A. Martínez, R. Martínez, F. J. Alfaro, and J. L. Sánchez, "A low-cost strategy to provide full qos support in advanced switching networks," *Journal of Systems Architecture*, vol. 53, no. 7, pp. 355–368, 2007.

[61] Connectx-6 dx the most secure and advanced cloud smartnic. NVIDIA. [Online]. Available: https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx/

[62] L. Foundation, "Data plane development kit (DPDK)," 2015. [Online]. Available: http://www.dpdk.org

[63] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin. (2018) Catboost: Unbiased boosting with categorical features. Red Hook, NY, USA.

[64] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[65] S. Bhattacharjee, R. Schmidt, K. Katsalis, C.-Y. Chang, T. Bauschert, and N. Nikaein, "Time-sensitive networking for 5g fronthaul network," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 2020, pp. 1–7.

[66] R. I.-T. G. Y. 1369.1, "Precision time protocol telecom profile for phase/time synchronization with full timing support from the network," 2020.

[67] NVIDIA, "Bluefield® smartnic for ethernet (2019)," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf.