OP4T: Bringing Advanced Network Packet Timestamping into the Field

Mohammed Hawari*†, Thomas Clausen†
*Cisco Systems, France
mhawari@cisco.com
†École Polytechnique, France
{mohammed.hawari,thomas.clausen}@polytechnique.edu

***Abstract*—Because it is very bursty, the microsecond-scale temporal behaviour of network traffic in data-centres is challenging to measure and understand. To bring observability into data-centre networks, this paper introduces the Open Platform for Programmable Precise Packet Timestamping (OP4T), a hardware architecture, targeting Field-Programmable Gateway Arrays (FPGAs), integrated into data-centre servers as a Smart Network Interface Card (SmartNIC), and flexible enough to enable advanced latency diagnosis.**

In this paper, OP4T is specified, and an open-source implementation of that architecture is proposed, targeting the NetFPGA SUME prototyping board. By leveraging the P4 programming language, and partial reconfiguration, that open-source implementation is experimentally shown to enable in-band, precise packet timestamping, without sacrificing the achievable throughput. As an illustration, OP4T is shown to be usable to measure fine-grained properties of a software packet forwarder, *e.g.,* packet batching.

I. Introduction

The accurate measurement of latency is a key tool for qualifying the performance of networked systems. Previous work has shown that traffic patterns in data-centre networks include packet bursts [1], [2], observed as a heavy-tailed distribution of packet interarrival times [3]. Bursts are responsible for an increased buffer occupation in packet switches, eventually leading to queuing and, if buffers are undersized, packet drops. The corresponding additional delays, even when they are in the microsecond scale, are in turn responsible for observable, application-level, performance impairments [4], [2]. Moreover, packet bursts in data-centre networks are transient, appear at time-scales in the order of a few dozen micro-seconds [5], and are difficult to detect by coarse measurements. That is different from traffic patterns, occurring in wide-area networks, and observable by methods such as tomographic inference [6], which are derived from coarse metrics,

To understand such transient network traffic patterns, and to diagnose transient latency spikes, instrumentation enabling accurate packet timestamping on selected flows is, therefore, crucial. Such instrumentation is already available as a part of *network testers*, *i.e.,* systems capable of generating predefined traffic patterns, and monitoring the latency introduced by networked Devices Under Test (DUTs). Despite the prior existence of network testers, both as commercial hardware appliances — *e.g.,* Ixia PerfectStorm or Spirent TestCenter, and as open source hardware designs — *e.g.,* the Open Source Network Tester (OSNT) [7] or FlueNT10G [8], the cost, programmability and/or performance of those solutions are subject to limitations, described in this paper. More fundamentally, those network testers are only designed to be used during the qualification phase of a DUT, and not *in situ*, *i.e.,* for understanding latency issues in a real deployment.

This paper goes beyond network testers by introducing the Open Platform for Programmable Precise Packet Timestamping (OP4T). While network testers are external to a DUT, and are responsible both for generating traffic patterns and for monitoring a temporal response, OP4T exposes a deliberately different semantic; OP4T belongs to the category of Smart Network Interface Card (SmartNIC) and exposes the same services as a regular network interface. Used in a data-centre server in place of a commodity Network Interface Card (NIC), OP4T enables in-band packet timestamping, with a minimal disruption of normal application operations.

The OP4T architecture is designed according to five guiding principles.

1) **Openness** Primarily targeted towards the research community, OP4T must be compatible with an affordable network prototyping Field-Programmable Gate Array (FPGA) board and, as much as possible, must reuse existing open-source hardware designs.
2) **Programmability** OP4T must allow programmable packet timestamping and payload alteration, to enable selecting the packet flows to monitor, and, potentially, the ones to alter with in-band timestamps. As such programmability must be accessible to network operators, not necessarily specialised in field programmable logic design, OP4T must provide a programming abstraction adapted to packet parsing, matching, and alteration, *i.e.,* equivalent to the one exposed by the P4 programming language [9].
3) **Precision** Destined to diagnose transient latency issues at small timescales, OP4T must be able to perform timestamping with a precision in the order of the microsecond at worst.
4) **Performance** When used to replace a regular server NIC, OP4T must not introduce any performance limitation in terms of achievable throughput or packet rate.
5) **Flexibility** Like most debugging, understanding transient latency spikes in a data-centre can be a complex, and

interactive process, *i.e.,* can require changes in the program defining packets to timestamp and alterations to perform. Therefore, those changes must be possible, with minimal disruption in network operations. For example, a full reprogrammation of the FPGA board is not acceptable, as it would require reloading the network interface driver. From a network operation perspective, this is highly disruptive.

The main contributions of this paper are (i) the design of OP4T as an architecture following those principles, (ii) the implementation of OP4T as an open-source hardware design, usable on the NetFPGA SUME board [10], along with a high-performance network interface driver, (iii) an experimental evaluation of the precision achievable by OP4T for a synthetic traffic pattern and a typical P4 program.

II. Related Work and Limitations

In this section, an overview of existing open-source solutions for packet timestamping is given. These solutions are analysed on two particular aspects: performance, and programmability. Other aspects and limitations of the packet timestamping capabilities of state-of-the-art network testers are already detailed in [8].

A. Performance

To analyse fine-grained latency variations, it is necessary to be able to timestamp, with high-precision, all the packets in a given stream. MoonGen [11] is an open source network tester, exploiting the Precision Time Protocol (PTP) support in commodity Network Interface Cards (NIC), to perform accurate packet timestamping. However, the Application Programming Interface (API) exposed by such a NIC exposes packet timestamps in an internal register, which is to be read and cleared by the driver each time a timestamp is to be retrieved. This severely limits the achievable packet rate in case all packets must be timestamped.

OSNT is capable of altering the received packets with a timestamp, inserted at a programmable position in the packet. The packet is then transmitted to the host server via Direct Memory Access (DMA), and the timestamps can be retrieved by parsing the received packets. However, preliminary experiments showed that the DMA core used by OSNT has insufficient performance to allow capturing a packet stream of 1.5 Gbps, which is only a fraction of the 10Gbps line-rate supported by the used NetFPGA-SUME board.

FlueNT10G timestamps packets in a way similar to OSNT, but rely on the Xilinx DMA/Bridge Subsystem for PCI Express v3.1 DMA core, which provides sufficient performance to saturate the PCI-Express bus, and therefore, allows capturing all the timestamped packets of a 10 Gbps stream.

B. Programmability

FlueNT10G offers a level of programmability by providing a software framework allowing network testing automation. However, the programmability of the hardware design itself is limited to specifying a list of MAC addresses, used to filter ingress packets. OSNT has more advanced filtering rules: it allows specifying a list of flows to timestamp, determined by IP-and-port-based packet matching rules.

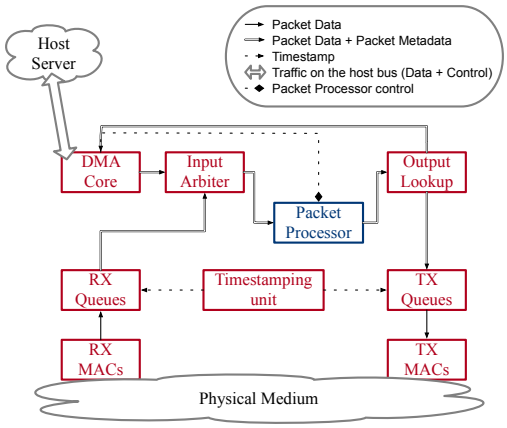

Figure 1. Abstract hardware architecture of OP4T. Red: elements from the static design. Blue: user-programmable packet processor.

III. Hardware Architecture

Figure 1 depicts the architecture of OP4T, independently from the underlying hardware target. In this section, the flow of a packet through this architecture is described, with a focus on timestamp acquisition, and the programmable packet processor.

A. Packet Flow

Figure 1 represents the different blocks traversed by each packet. Along with its data, packet *metadata* is transported between the different blocks. Metadata includes information related to the packet, and which is to be shared across blocks, *e.g.,* the source and destination physical ports.

As OP4T mimics the behaviour of a network interface, a packet entering the design can originate either from the host server, or from the physical medium. In the first case, packet data and metadata are retrieved from the host memory by the DMA core of figure 1. In the second case, the packet is received over one of the physical mediums connected to OP4T, by the corresponding Reception Medium Access Control core (RX MAC). The packet data is then enqueued into a reception queue (RX Queue), along with metadata, generated on-the-fly. In both cases, packet metadata includes the *source* of the packet, *i.e.,* whether it was generated by the host server, or the identity of the reception physical medium.

Regardless of its source, the packet is then transmitted to an input arbiter with multiple packet inputs and one output. The input arbiter selects a packet from one of its inputs, and transmits it to the packet processor. It alters the packet data and metadata by executing user-specified operations. In particular, a *destination* is added to the metadata. The packet is finally transmitted to the output lookup block, which, depending on the destination, routes it either to the DMA core for transmission to the host server, or to one of the transmission queues (TX Queues), for transmission over one the connected physical mediums.

B. Timestamp Acquisition

Timestamping is performed in two places in the design: at the ingress of the RX Queues, and at the egress of the TX Queues. Timestamping as closely to the MACs as possible eliminates the jitter introduced by, *e.g.,* the different sources contending at the input arbiter, or by the packet processor.

At the RX Queue, a timestamp is generated for each incoming packet, and inserted in the packet metadata for later consumption by the packet processor. However, that is not applicable at the TX Queue, as metadata are lost upon packet transmission over the physical medium. Therefore, a timestamp acquired at the TX Queue must be inserted in the packet data. To avoid altering all the packets flowing through the design, the packet metadata includes a flag, set by the packet processor, and indicating whether a timestamp should be added by the TX Queue. Moreover, if that flag is set, the packet metadata must also include the data offset where the timestamp should be inserted.

C. Reconfigurable Packet Processor

The packet processor contains user-defined logic, performing data and metadata alteration. That logic is expected to set the destination information in the metadata, so that the output lookup block can route the packet. Typically, to mimic the behaviour of a NIC, the user-defined logic should read the source from the metadata, and route packets originating from a physical medium towards the host, and those originating from the host to the corresponding physical medium.

Runtime flexibility is brought to the packet processor by *control-plane*, and by *partial logic reconfigurability*. As represented on figure 1, control-plane enables the host server to push stateful information into the packet processor at runtime, *e.g.,* traffic matching rules, specifying packets to be altered with timestamps. Partial logic reconfigurability is a feature of some FPGAs, enabling updates of part of the programmed logic, without erasing and resetting the whole design. In OP4T, the packet processor is partially reconfigurable, enabling live updates, with minimal disruption from the perspective of the host server. When performing live latency debugging, this allow a network operator to push a packet processor logic, specifically tailored to the current debugging scenario, without a full reset of what appears as a network interface to the host server.

In the remainder of this paper, the term *static design* designates all the components of figure 1 that are not reconfigurable at runtime, *i.e.,* all the elements at the exclusion of the packet processor.

IV. Implementation

The hardware architecture described in section III was implemented on the NetFPGA SUME FPGA board [10], which is, at the time of writing this paper, the de facto standard prototyping platform in the network research community. As much as possible, this implementation also relies on open-source Intellectual Property (IP) cores, as detailed in the following.

A. Overview

The presented implementation of OP4T is derived from OSNT-SUME, *i.e.,* the adaptation of OSNT to the NetFPGA-SUME board. Specifically, the RX and TX MACs, timestamping unit, input arbiter and output lookup block from figure 1 are reused from OSNT-SUME. The RX Queues and TX Queues blocks are derived from the ones used in OSNT-SUME, which were modified to implement timestamping as described in section III-B. Following the OSNT-SUME implementation, the block interconnections from figure 1 are implemented by AXI4-Stream buses for packet data and metadata transport, an AXI4-Lite bus for the control-plane, and the PCI Express bus for the interface with the host server.

Because of its performance limitations, the DMA core included in OSNT-SUME and NetFPGA-SUME was replaced with a custom implementation, which consists of a wrapper around the Xilinx DMA/Bridge Subsystem for PCI Express (XDMA) IP core, provided as part of the Xilinx toolchain.

Finally, the P4-NetFPGA workflow and source code [12] are partially reused and adapted to allow the creation of custom packet processors in P4.

B. DMA Core integration

The proposed implementation of OP4T uses the XDMA IP core to provide high-performance packet transmission and reception over PCI-Express (PCIe). Specifically, XDMA provides Card To Host (C2H) and Host To Card (H2C) *channels*. Those appear to the host as queues of read (for a C2H channel) or write (for a H2C channel) *DMA operation descriptors*. Each descriptor is first enqueued by the software, then dequeued by the XDMA hardware, which finally executes the associated read/write DMA operation. On the hardware design, channels appear as AXI4-Stream (for the data-plane) or AXI4-Lite (for the control plane) input (for a C2H channel) or output (for a H2C channel) ports belonging to the XDMA core.

Because the host is to be exposed the semantic of a network interface, each received packet is mapped to a read DMA operation, and each transmitted packet, to a write DMA operation. That is performed in the proposed OP4T implementation by a Data Plane Development Kit (DPDK) Poll-Mode Driver, communicating with XDMA over PCIe, and translating DMA operations into packets.

C. P4 Packet Processor and Partial Reconfiguration

To facilitate the implementation of custom user-logic for the packet processor, the proposed design extends the Xilinx Vivado Partial Reconfiguration flow [13], as depicted in figure 2. First, the static design is synthetised **(a)**, into a netlist. Then, a first flavor of the user-logic, denoted by **packet processor 0** and written in P4, is translated into Register-Transfer Level (RTL) code **(b)** by the Xilinx P4-SDNet and Xilinx SDNet compilers[1], following the P4-NetFPGA workflow [12]. The generated RTL

[1]This toolchain is described by documentation available at https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1252-p4-sdnet.pdf and https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1012-sdnet-packet-processor.pdf

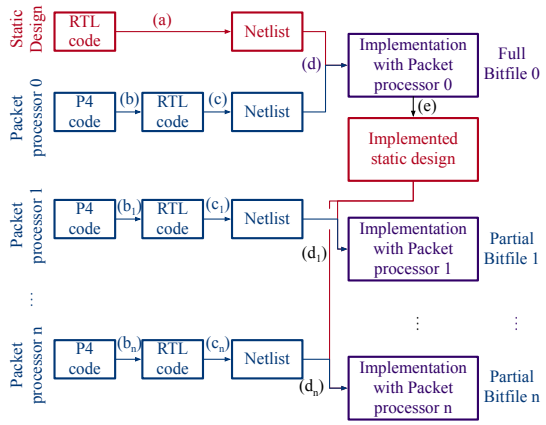

Figure 2. Vivado Partial Reconfiguration flow applied to implementing P4 packet processors for OP4T. Red: elements of the static design. Blue: elements of the packet processor. Purple: output products combining elements from the static design and from the packet processor.

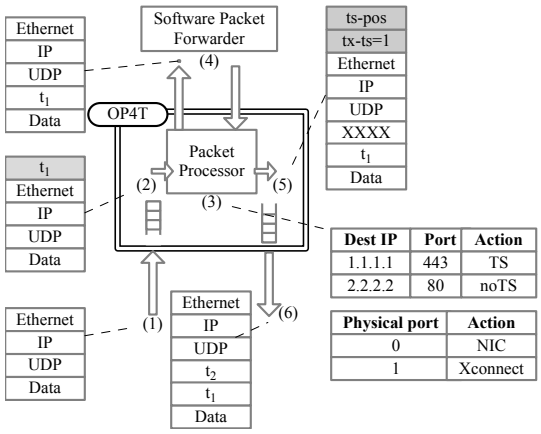

Figure 3. Packet Flow when testing a software switch. Greyed information is metadata transported with a packet. t_1 is the timestamp sample upon packet reception from the network. tx-ts is the flag indicating that the TX Queue must add a timestamp upon transmission, at the position indicated by ts-pos.

code is then synthetised **(c)** into a netlist, which is combined with the netlist of the static design. The resulting netlist is placed and routed **(d)**, finally forming an implementation of OP4T with Packet Processor 0 (OP4T-PP0).

To enable partial reconfiguration of the packet processor, *i.e.,* replacement of packet processor 0 with another P4-based user logic, the implementation of OP4T-PP0 is first stripped from all placement and routing information related to packet processor 0 **(e)**, resulting in the implemented static design. Then, another P4 code, denoted by **packet processor 1**, is translated into RTL code **(b_1)**, synthetised **(c_1)**, and combined with the implemented static design **(d_1)** for placement and routing. The result of that operation is an implementation of OP4T with Packet Processor 1 (OP4T-PP1), fully compatible with the one of OP4T-PP0, *i.e.,* placement and routing of the elements of the static design are identical in both implementations. Therefore, partial reconfiguration is possible by only reprogramming, at runtime, the FPGA area corresponding to the packet processor. This operation can be repeated for as many additional P4-based user logics as needed, as shown in figure 2.

Placement and routing of OP4T with additional packet processors are constrained by the placement and routing of OP4T-PP0 **(d)**, which are constrained by the complexity of packet processor 0 itself. Therefore, when using the workflow of figure 2 for implementing multiple configurations OP4T-PP0, OP4T-PP1, ..., OP4T-PPn, packet processor 0 should be chosen so that OP4T-PP0 is the most challenging configuration for the placement and routing engines.

D. Discussion

The workflow detailed in section IV-C enables implementing multiple versions of OP4T, each with a different packet processor. Moreover, partial reconfiguration enables switching from one version to the other, without disrupting the static design, thus, with neither resetting the DMA core, nor impacting the network interface exposed to the host server. However, two main difficulties arise when implementing partial reconfiguration.

Firstly, it requires *floorplanning*, *i.e.,* the FPGA physical resources allocated to the static design and to the packet processor must be determined prior to placement and routing. Although automated tools can assist floorplanning [14], [15], [16], it still requires prior knowledge of the resource utilisation induced by the packet processor. Therefore, once floorplanning is performed, the thereby provisioned resources limit the complexity of any future packet processor.

Secondly, choosing packet processor 0 so that OP4T-PP0 is the most challenging design to place and route implies prior knowledge of all future packet processors, which is not necessarily possible in the scenarios targeted by OP4T, *e.g.,* generating an ad hoc packet processor to diagnose a given latency issue, in a production data-centre, without fully reprogramming the FPGA. The proposed implementation avoids that issue by explicitly registering all the inputs and outputs between the static design and the packet processor, at the cost of increased latency.

V. Case Study: OP4T for Software Switch Testing

In this section, the OP4T implementation from section IV is used in a latency evaluation scenario, different from network testing: evaluating the latency introduced by a software packet forwarder itself, *i.e.,* only the latency introduced by packet processing and PCIe-based DMA. To that end, a packet processor is specified, inserted in the OP4T design, eventually forming the OP4T for Software Switch Testing (OP4T-SST) configuration.

A. Scenario

When evaluating the latency of a packet forwarder (DUT), a network tester generates a packet stream (testing stream), transmits it to the DUT and monitors the packets forwarded by the latter. The Round Trip Time (RTT) obtained by comparing transmission and reception timestamps at the network tester provides an evaluation of the latency of the DUT. While that measurement is impacted by the packet delay variations due to the network path, those can be considered as negligible in

a controlled infrastructure (part of a network testing setup). Consequently, the obtained measurements are precise.

However, in a real deployment hosted in a data centre running production applications, network-path induced packet delay variation incurred by testing traffic is non-negligible, due to, *e.g.,* congestion and queueing occurring in the data-centre switching fabric. To accurately evaluate the latency of a software packet forwarder — typically, a Virtual Network Function — deployed in such a data-centre, it is necessary to timestamp packets upon reception and transmission, at the network interface. If the latter is based on OP4T, such timestamping can be implemented by a custom packet processor. In this case study, the testing stream to be selected by OP4T for timestamping, is a User Datagram Protocol (UDP) stream.

B. OP4T-SST Packet Processor

The behaviour of the packet processor used in OP4T-SST is represented in figure 3. When a packet is received (1), OP4T generates a timestamp t_1, inserted in the packet metadata (2). The packet processor then parses the packet's headers, and performs a table lookup (3). The destination Internet Protocol (IP) address is matched against a table (created by the host server, through the control plane), to determine whether the packet belongs to a stream of interest to be timestamped. This lookup can either yield a TimeStamp (TS) or a do not TimeStamp (noTS) action. If the TS action is matched, then, t_1 is inserted between the UDP header and the data. Otherwise the packet is not edited. Then, the processed packet is sent to the host through the DMA core (4). As a Software Packet Forwarder is running on the host, the packet is sent back to OP4T, and, the packet processor performs the same table lookup as before. If the TS action is matched, the packet is edited to provision zero-ed bytes immediately after the UDP header, and the packet metadata is edited, so that, upon transmission, the TX Queue block replaces the provisioned bytes with a transmission timestamp t_2 (5). Finally, the resulting packet contains, just after the UDP header, two timestamps t_2 and t_1, whose difference evaluates the latency introduced by the Software Packet Forwarder.

C. Precision and Cross-connect

With the previously described flow, the measured latency includes the delay introduced by the packet processor itself. Due to internal pipelining, this delay may not be constant, impacting measurement precision.

To evaluate the loss thereof, the OP4T-SST packet processor has a cross-connect (Xconnect) feature. When a packet is received from the network, it is not necessarily sent to the DMA core. Instead, a lookup is performed on another table, different from the one mentioned in section V-B , as shown in figure 3. If the Xconnect action is matched in this table, the packet is directly transmitted back to the network, without going through the Software Packet Forwarder. Moreover, in that case, if the packet is determined to match the TS action, t_2 and t_1 are both appended after the UDP header, thus, providing an evaluation of the base latency introduced by the packet processor.

VI. Evaluation

In this section, a quantitative evaluation of OP4T-SST is provided.

A. Experimental Setup

The experimental study conducted in the following aims at evaluating the precision of OP4T-SST, as well as exploring how features of the testing traffic impact the latency necessarily introduced by the design.

The software packet forwarder under test is the Vector Packet Processor (VPP) [17]. To obtain precise and reproducible results, this evaluation focuses on the latency experienced by constant-rate packet streams with a stable period. Contrary to state-of-the-art network testers, OP4T does not integrate any packet generator. Consequently, the method developed in MoonGen is used for generating a testing stream with a stable period. That method consists of transmitting, at line-rate, alternately one packet from the testing stream, and a certain number of packets, crafted so as to be dropped before reaching the receiver (*e.g.,* with a bad Cyclic Redundancy Check(CRC)). That last number of packets determines the period achieved by the generator.

The experimental setup consists of a packet generator, a packet monitor, and a server running VPP and hosting a network interface implemented as a NetFPGA-SUME board programmed with OP4T-SST. A testing stream is transmitted by the generator, traverses the server through OP4T (as depicted in figure 3), and is finally received by the packet monitor. The latter then estimates the latency incurred by each packet by computing the difference between the two timestamps t_1 and t_2.

Finally, OP4T-SST and/or VPP must be configured to forward the stream received from the packet generator to the packet monitor. Four ways were experimentally implemented to achieve that goal: (i) using the Xconnect feature of OP4T-SST, effectively bypassing VPP and yielding a baseline latency estimation, (ii) using the layer-2 patch feature of VPP (iii) using the layer-2 cross-connect feature of VPP (iv) using VPP as a layer-3 packet switch with an appropriately configured routing table. As those three last VPP-based configurations rely on code paths of increasing complexity, comparing the measured latencies in those configuration evaluates the ability of OP4T-SST to detect fine-grained software behaviour.

B. Results

Figure 4 summarises the obtained results. The baseline experiment shows that, the latency incurred by traversing OP4T has a negligible standard deviation (around 4 nanoseconds). This number is difficult to improve, as the used implementation is clocked at 250MHz, *i.e.,* with a 4 nanoseconds period. Moreover, the measured baseline latency only depends on the packet sizes, not on the packet rate. Dependency on packet sizes is explained by store-and-forward packet transmissions at

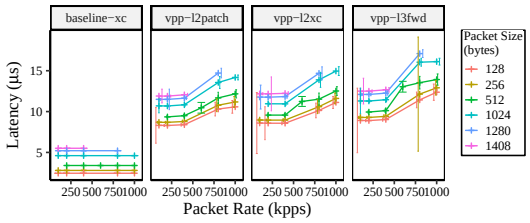

Figure 4. Measured average latency experienced by testing streams of various packet rates and sizes. The server traversed by the stream is configured (from left to right): (i) with the OP4T-SST Xconnect feature, (ii) with the layer-2 patch VPP feature, (iii) with the layer-2 cross-connect VPP feature, (iv) with VPP configured as a layer 3 packet switch. Error bars are centred around the average and their height is twice the standard deviation of the latency.

the ingress and egress of the hardware design, which necessarily add an extra-latency proportional to the packet size.

The results obtained in the three VPP-based configurations realistically illustrate the internal behaviour of VPP. Firstly, the latency increases with the complexity of the involved code, with a clear difference between layer-3 forwarding and layer-2 cross-connect. Secondly, the standard deviation also increases with the complexity of the code, which is explained by an increased number of sources of packet processing time variations (cache-misses, software or hardware preemption). Finally, in a given configuration and for a given packet size, latency is constant when increasing the packet rate, until a clear threshold (400000 packets per second), beyond which latency is linear. This is an experimental verification of batched packet processing occurring in VPP.

VII. Conclusion

OP4T specifies an open programmable architecture, capable of high-precision packet timestamping, in situ, *i.e.,* deployed in a data-centre. To achieve that goal, OP4T is designed to be usable simultaneously as a network interface, transmitting and receiving production traffic, and as a partially reprogrammable packet timestamp acquisition device, altering selected packets by the adjunction of reception and/or transmission timestamps. The programmability of OP4T is key to debugging complex latency issues, as it brings the ability to interactively refine the packet timestamping logic, without disrupting the exposed network interface.

This architecture was implemented on the NetFPGA-SUME board, relying on open source IP cores derived from the NetFPGA-SUME, P4-NetFPGA, and OSNT-SUME projects. Specifically, programmability was achieved by the joint use of the P4 programming language, and partial logic reconfigurability provided by modern FPGAs. The obtained open-source implementation was shown to achieve timestamping with a precision in the order of a single clock cycle, and was shown to be precise enough to measure fine-grained properties of a software packet forwarder such as VPP, which is a synthetic example of data-centre application

References

[1] J. Woodruff, A. W. Moore, and N. Zilberman, "Measuring Burstiness in Data Center Applications," in *Proceedings of the 2019 Workshop on Buffer Sizing*. ACM, pp. 1–6. [Online]. Available: http://dl.acm.org/doi/10.1145/3375235.3375240

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," vol. 40, no. 4, pp. 63–74. [Online]. Available: https://doi.org/10.1145/1851275.1851192

[3] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. Association for Computing Machinery, pp. 267–280. [Online]. Available: https://doi.org/10.1145/1879141.1879175

[4] D. A. Popescu, N. Zilberman, and A. W. Moore, "Characterizing the impact of network latency on cloud-based applications' performance." [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-914.html

[5] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17. Association for Computing Machinery, pp. 78–85. [Online]. Available: https://doi.org/10.1145/3131365.3131375

[6] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '09. Association for Computing Machinery, pp. 202–208. [Online]. Available: https://doi.org/10.1145/1644893.1644918

[7] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. Mckeown, N. Feamster, B. Felderman, M. Blott, A. W. Moore, and P. Owezarski, "OSNT: Open source network tester," vol. 28, no. 5, pp. 6–12.

[8] A. Oeldemann, T. Wild, and A. Herkersdorf, "FlueNT10G: A Programmable FPGA-based Network Tester for Multi-10-Gigabit Ethernet," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 178–1787.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," vol. 44, no. 3, pp. 87–95. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[10] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," vol. 34, no. 5, pp. 32–41. [Online]. Available: http://ieeexplore.ieee.org/document/6866035/

[11] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC '15. Association for Computing Machinery, pp. 275–287. [Online]. Available: https://doi.org/10.1145/2815675.2815692

[12] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4->NetFPGA Workflow for Line-Rate Packet Processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. Association for Computing Machinery, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3289602.3293924

[13] K. Vipin and S. A. Fahmy, "FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications." [Online]. Available: https://doi.org/10.1145/3193827

[14] P. Banerjee, M. Sangtani, and S. Sur-Kolay, "Floorplanning for Partially Reconfigurable FPGAs," vol. 30, no. 1, pp. 8–17, conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[15] M. Rabozzi, G. C. Durelli, A. Miele, J. Lillis, and M. D. Santambrogio, "Floorplanning Automation for Partial-Reconfigurable FPGAs via Feasible Placements Generation," vol. 25, no. 1, pp. 151–164, conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[16] M. Rabozzi, J. Lillis, and M. D. Santambrogio, "Floorplanning for Partially-Reconfigurable FPGA Systems via Mixed-Integer Linear Programming," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 186–193.

[17] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi, "High-Speed Software Data Plane via Vectorized Packet Processing," vol. 56, no. 12, pp. 97–103, conference Name: IEEE Communications Magazine.