Deliberately Congesting a Switch for Better Network Functions Performance

Mariano Scazzariello
KTH Royal Institute of Technology

Tommaso Caiazzi
Roma Tre University

Marco Chiesa
KTH Royal Institute of Technology

***Abstract*—Traditional wisdom suggests maintaining minimal occupancy in the port queues of network devices to prevent packet delays or drops *en route* to their destination. In this paper, however, we explore the unconventional idea of deliberately *congesting* the queues of a network device to *enhance* the performance of a Network Function (NF) deployment. The key intuition behind this approach is to utilize the existing memory available in the switch queues to store packet payloads while their headers are processed on an external NF processor. We present two techniques for congesting a port on a switch: i) *self-clocking* packet recirculation, which recirculates packets within the switch to automatically achieve the correct queuing delay, and ii) a proportional *controller* using multicast forwarding, which adjusts the rate of packet forwarding based on the level of congestion. We evaluate our approaches both in simulations and a prototype.**

I. Introduction

Network Functions (NFs) are an essential part of today's networks. Operators have increasingly deployed NF processors on external fully programmable devices (*e.g.,* CPU, FPGA) as opposed to fixed-function network switches. This is an effective way to support arbitrary packet processing pipelines that need to be adapted over time to new use cases and requirements, something that is challenging to support with fixed-function ASIC switches [1].

Recent approaches to deploy NFs on fully programmable devices advocate for *splitting* packet headers from packet payloads in the network (on ASIC switches) and to process only the packet headers on the external programmable CPU or FPGA device while temporarily storing the payloads somewhere in the network [2]–[4]. Headers and payloads are then recombined to form the processed packet. Processing only headers on the NF processor results in a 5-10× throughput increase compared to traditional solutions [2].

Storing packet payloads, however, presents a considerably greater challenge than it might seem. ASIC switches cannot today store full payloads in a programmatic manner [2]. For this reason, recent approaches [2], [3] store payloads on devices external to the ASIC switch (*e.g.,* RDMA-enabled servers, NICs), which entails high bandwidth overheads on the ASIC switch to store/fetch payloads.

Despite ASIC switches not offering a programmatically accessible memory for packet storage, we observe that there is an inherent type of memory performing this task on a switch: the buffer memory for each port's queue. This

979-8-3503-5171-2/24/$31.00 ©2024 IEEE

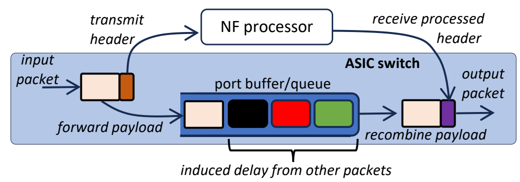


Fig. 1. TurboSwitch congests the queue of the output port to temporarily store the payload of a packet. The corresponding header gets processed on an external NF entity.

memory stores packets when congestion on a port arises and drains the queue at the speed of the output port. Our intuition is that one could store payloads in any port queues while the external NF processor handles the corresponding headers. In this paper, we show that if the queue occupancy is large enough, by the time the payload leaves the queue, the NF processor will have already sent back the new header, which can be recombined with the payload and forwarded out of the switch without incurring any bandwidth overheads.

We present TurboSwitch, a system design that deliberately congests the queues of a switch to store there payloads while their corresponding headers are being processed on an external NF (see Fig. 1). The key challenge is keeping queues filled with the correct number of packets. We present two approaches to control the queue occupancy. The first approach is based on recirculating some payloads of the packets through the switch (*i.e.,* processing them more than once). This approach is particularly appealing as it *self-clocks* to the correct queue size without any sophisticated or complex packet processing logic. The second approach relies on controlling the queue by multicasting each packet a certain number of times, which artificially increases the load on an output port and potentially builds up a queue.

We perform a preliminary evaluation of TurboSwitch in simulations and deploy a proof-of-concept on a programmable ASIC to understand the potential challenges of realizing TurboSwitch. Our initial results indicate that the multicast-based approach is the most performing approach (because it avoids wasting some bandwidth by recirculating packets) yet it is more challenging to realize in practice (because of the need to carefully generate the correct number of multicasted copies of a packet).

II. Storing payloads

We now discuss different architectural choices for storing payloads while processing their corresponding headers.

Deploying NFs entirely on ASIC switches. Ideally, an NF processor would run entirely on the data-plane of an ASIC switch to fully utilize the packet processing throughput of the switch (see first row of Table I). The packet header would be processed through the pipe logic while the payload would be stored on a fast dedicated SRAM. Unfortunately, today's existing ASIC switches may not be able to support advanced stateful NFs as these either require excessive memory to store the state or their logic is too complex for running on ASIC [2] as in the case of TEA [5].

Deploying NFs on external NF processors. To implement complex NFs, today's state-of-the-art systems offload simple parts of NFs to ASIC switches and run more complex parts on external NF processors (second row of the table and below). For instance, Tiara [6] (2nd row) uses half of the ports on the switch to receive/send traffic and the other half to forward traffic to/from an external FPGA-based NF processor. Compared to the ideal ASIC-based deployment, the throughput is halved due to sending back and forth full-sized packets to the external NF processor. Moreover, external processors also need to receive payloads that are unused even if these NFs do not need to process them (as it is the case for so-called shallow NFs [3], [4]). Even just receiving payloads has a significant impact on the performance of NFs (*e.g.,* by cache pollution [7]).

Splitting packet headers from payloads. For the above reasons, three recent systems advocate for only processing packet headers on external NF processors (for better CPU cache hit-ratio and utilization) and storing payloads in various locations: on the NIC with nicmem [3] (3rd row), and on RDMA servers with Ribosome [2] (4th row), or on the switch with PayloadPark [4] (5th row). We first note that the first two approaches also suffer from throughput degradation, since the switch stores the payloads on external devices (on the external server NIC or memory). Unfortunately, even PayloadPark, which aims to store payloads on the switch, suffers from the same problem as it can only store 160 B of a packet payload, which means that the switch still has to transmit a large fraction of the payload bits to external devices [2]. An alternative approach would be to attach long fibers on the switch to delay payloads (6th row), *e.g.*, a 4-km optical fiber has a latency of $20\mu s$. There are three disadvantages: *i)* throughput is still halved as the fibers can only be used to store payloads, *ii)* fibers (and transceivers) are an additional cost, and *iii)* fibers only add a static delay.

Our goal: deploying advanced NFs at full speed. In this paper (last row), we embark upon the design study of storing payloads entirely on the switch, thus achieving full throughput. By only transmitting headers to external NF processors, such an approach would support high CPU efficiency and advanced NF functions.

III. A QUEUE-BASED PACKET STORAGE

We envision a new approach for increasing the throughput of widely-deployed *shallow* NF packet processors that only operate on packet headers, *e.g.,* Layer-4 load balancers, packet schedulers, firewalls. In our design, we process headers on an external NF processor, and we store packet payloads on the per-port queues of the ASIC switch. These queues buffer packets whenever the rate of incoming packets directed towards an outgoing port exceeded the rate at which packets could be forwarded on that port, *e.g.*, during a traffic incast, congestion events, or differences in the input/output link speeds. We present two approaches to store payloads in a queue: one based on recirculating packets (§III-A) and one based on multicasting packets (§III-B).

Processing	Network Throughput	CPU Efficiency	Advanced NFs?
Only on ASIC	full ✓	no CPU ✓	no ✗
Non-splitting NF (*e.g.,* Tiara [6])	half ✗	low ✗	yes ✓
nicmem [3]	half ✗	medium ✗	yes ✓
Ribosome [2]	half ✗	high ✓	yes ✓
PayloadPark [4]	half ✗	low ✗	yes ✓
Long fibers	half ✗	high ✓	yes ✓
TURBOSWITCH	full ✓	high ✓	yes ✓

TABLE I
QUALITATIVE COMPARISON AMONG EXISTING SYSTEMS.

A. A Self-Clocked System with Recirculation

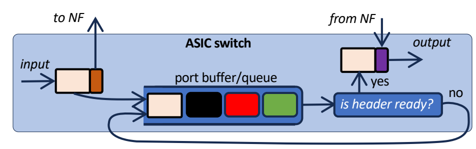


Fig. 2. The recirculation-based approach.

Our first design entails recirculating payloads within a switch until the external NF processor produces the new header for that payload. Today's switches allow operators to configure (a subset of) their ports in a so-called *loopback* mode. When a switch forwards a packet to a port in loopback mode, the packet will re-enter the switch from the same exact port and will be re-processed, *i.e.*, the packet recirculates.

Key intuition: Self-clocked queues. When payloads recirculates through the switch, they ultimately generate an incast on a port on the switch, where a queue builds up. Interestingly, the queue occupancy automatically *self-clocks* to a specific value that will make payloads wait in the queue until the NF produces the header. We describe the above intuition in more details in the following.

Description of the mechanism. In this approach, the switch recirculates each payload of a packet until the NF processor produces the new header for the payload, as shown in Fig. 2. For simplicity, we assume that there is a single input and output port. We configure a third port on the switch in loopback mode to recirculate packets. When a packet enters the switch, the switch splits the packet into a header and a payload. The switch sends the header *hdr* to a port that is connected to an external packet processor, which computes a new header hdr' and sends it back to the switch. This operation typically takes any time between 5 and $500\mu s$ depending on the complexity of the NF implemented [6] [8]. During this time, the switch keeps recirculating the payload through the loopback port. Every time the switch extracts a payload from the queue, it checks whether the NF has transmitted the new header for that payload.

Example. Consider an example with 25 Gbps of input traffic and a 100-Gbps loopback (recirculation) port. For simplicity, assume the only packet processing latency on the switch is due to queuing delays. By recirculating payloads (potentially multiple times), the switch increases the amount of traffic on the loopback port. When the loopback port receives a 100 Gbps traffic incast (*i.e.*, 25 Gbps from normal input traffic and 75 Gbps of recirculated traffic), a queue starts to build up on the loopback port. The switch indirectly ends up exploiting this queue memory to temporarily store payloads while waiting for their headers. If the NF takes 40μs to produce a header and each packet traverses the queue four times (by recirculating three times), the queue will grow so that the queuing time of a payload will be roughly 10μs. Note that if the queue were to grow to 20μs, then packets only traverse the queue twice, which means packets would recirculate only once. In this case, however, the traffic generated on the loopback port would only be 50 Gbps (25 Gbps of input and 25 Gbps recirculated). This situation would immediately lead to draining the queue, which will increase the number of recirculations, and consequently the traffic on the loopback port (so that the queue builds up again on the loopback port). To conclude, the queuing time *self-clocks* to a specific value that depends on the external NF processor latency and the average number of payload recirculations.

Modelling the steady-state system. We now build a simple analytical model of the self-clocked system to reason about the queuing memory requirements. We make some assumptions: *i)* the network function takes exactly NF_{lat} time units to process a packet including network latencies from/to the switch, *ii)* the packet processing latency within a switch (without including the queuing time) plus the recirculation link latency is sw_{lat} time units, and *iii)* there is a single port queue on the switch to buffer packets. We omit all proofs and will publish them as a technical report.

We first observe that, at steady state, recirculating the input packets several times amplifies the input traffic rate. We denote by in_{tput} the input throughput (without recirculations) and by n_{recirc} the average number of recirculations per packet. The effective input rate on the loopback port is therefore $in_{tput}(n_{recirc}+1)$. Now, to build a stable queue on the loopback port, the input rate on the queue must match the throughput sw_{tput} of the loopback port, that is, $in_{tput}(n_{recirc}+1) = sw_{tput}$. The following fact follows.

Fact 1. *The average number of recirculations for each packet only depends on the input rate and loopback port throughput. More specifically:*

$$n_{recirc} = \frac{sw_{tput} - in_{tput}}{in_{tput}} \tag{1}$$

Note again that the above number is the *average* number of packet recirculations. It is not guaranteed that each packet recirculates exactly the same number of times. Intuitively, we can expect a distribution around the average value.

We now lower bound the average queue size $queue_{size}$ on the loopback port to estimate the queue size requirements. We observe that it takes $\frac{queue_{size}}{sw_{tput}}$ time units for a packet to go through the queue. Since a packet will stay in the system for at least NF_{lat} time units, we have that:

$$\underbrace{\frac{queue_{size}}{sw_{tput}}}_{\substack{\text{queueing time} \\ \text{w/o recirc. latency}}} + n_{recirc} \underbrace{\left(sw_{lat} + \frac{queue_{size}}{sw_{tput}}\right)}_{\substack{\text{recirculation latency} \\ \text{including queueing time}}} > NF_{lat}$$

By Eq. 1, the above inequality leads to the following fact.

Fact 2. *The queue size increases with an increasing NF latency or input rate, while it decreases with an increasing packet processing latency on the switch. More formally:*

$$queue_{size} > NF_{lat} in_{tput} - sw_{lat}(sw_{tput} - in_{tput})$$

Note that the lower bound of $queue_{size}$ is positive only when $sw_{lat} \leq \frac{in_{tput} \cdot NF_{lat}}{sw_{tput} - in_{tput}}$. When this condition does not hold, the switch latency sw_{lat} (which does not include recirculation latency) is so high that recirculating a packet $recirc$ times already takes more than NF_{lat} time units, which does then not require building any queue.

With a simple analysis (omitted), we can also upper bound the maximum queue size and packet processing latency.

Theorem 1. *The maximum queue occupancy is less than* $2 \times NF_{lat} sw_{tput}$ *packets.*

Corollary 1. *The maximum packet processing latency is* $3 \times NF_{lat}$ *time units.*

Putting the above formulas into perspective. There are a few important takeaways from the above analysis. First, we observe that the number of times a packet recirculates does *not* depend on the NF packet processing latency but only on the ratio between the input traffic rate and port throughput. We observe that the maximum queue size arises when packets do not recirculate, since packets traverse the queue only once and they must immediately find their header. Consider a 1.6-Tbps pipe on a switch and a network function latency of $20\,\mu s$. In this case, the maximum queue occupancy is 8000 using 1000B-sized packets based on Theorem 1. This is in line with the buffer memory available on a 1.6 Tbps pipe of a datacenter switch.

Handling packet re-ordering. Each packet in TURBOSWITCH may recirculate a different number of times. As such, packets belonging to the same flow may be forwarded out of order. One way of guaranteeing that packets leave the switch in the correct order is to number packets when they enter a switch (possibly partitioning flows with a hash function and assigning distinct counters to these different partitions). The cost of guaranteeing packet reordering is a potential higher number of packet recirculations and, consequently, lower throughput. The magnitude of the throughput reduction depends on the number of flows and the number of partitions. We evaluate these aspects in §IV.

B. A Controller System with Multicast

We now discuss an *alternative* design of TURBOSWITCH that relies on packet multicasting capabilities.

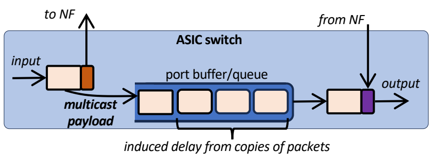


Fig. 3. The multicast-based approach.

Key-idea: using a control system to build a queue. In this approach, the switch multicasts copies of the incoming packets to deliberately build up a queue on the port, as shown in Fig. 3. The switch must "guess" how many copies of a packet must be generated in order to keep the queue occupancy at the "correct" value, *i.e.*, a number of copies that will delay a payload by more than the NF packet processing latency. For instance, when the input rate is halved, the switch should double the number of generated multicast packet copies in order to keep the queue at the same value. We envision a switch relying on a control system that monitors the current queue size and reacts to the error between that value and the target value using, for instance, a Proportional-Integral-Derivative (PID) controller [9].

Example with a proportional controller. We now discuss a simple example based on a PID controller that only uses the proportional control part to adjust the queue size by generating a different number of copies of an input packet entirely in the data plane. Consider an NF that takes 20μs to process packets. The queue size should be kept at least at 2k packets to delay packets by 20μs. The switch periodically obtains a measurement of the queue size (with some delay). It then computes a queue size estimate based on the delay δ in receiving the measurement, which can be obtained with some accuracy level using hardware timestamps. The adjustment consists of both increasing/reducing the queue size: an increase based on the number of multicasted packets generated in the last δ time units and a decrease based on the estimated number of packets dequeued from the queue port in the last δ time units.

C. Challenges

A key question about TURBOSWITCH is how to map the packet processing logic onto the data-processing pipelines of an ASIC switch. The purpose of the following discussion is to only shed light on the potential challenges of a queue-based approach without proposing an ultimate solution.

Dynamically changing the output port. A key assumption in §III is that a switch can modify the selected output port after dequeueing a packet from the per-port queue. This happens when the selection of the output port depends on the external NF processor and cannot be known by the switch when enqueueing a packet in a per-port queue. The simple solution today is to recirculate a packet once more and select the correct output port when re-processing the packet. We, however, note that in many NF scenarios, the responsibility of determining the output port belongs to the switch and *not* to the external NF. This is, for instance, the case for traditional datacenter designs where Internet traffic enters from a Point-of-Delivery (PoD) [10]. The Internet-facing PoD deploys a variety of NFs including security firewalls, proxies, L4 load balancers, NATs, and more. Some of these NFs are responsible for computing the server that should receive a packet. But, all the output ports on a PoD switch can be used to reach any server in today's datacenter topology (*e.g.,* Clos-based). The output port selection can therefore be performed at the switch level by using any of existing mechanisms (*e.g.,* ECMP, WCMP [11], Flowlet [12], NDP [13]). We, therefore, believe that the multicast approach would not need to recirculate packets when deployed in such scenarios.

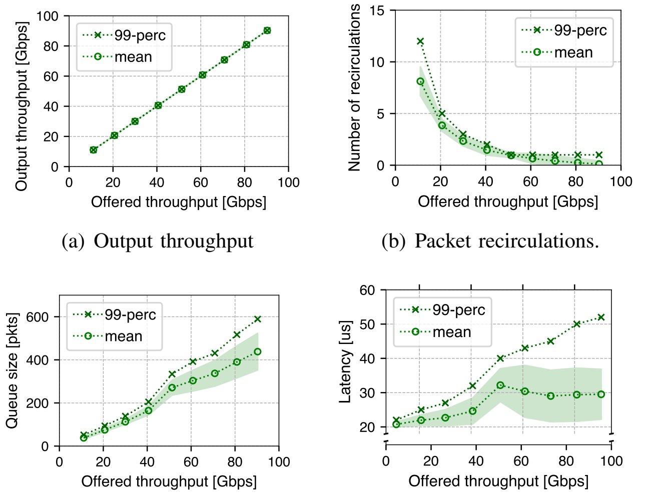
Dynamically re-assigning recirculation ports. In the recirculation-based approach, the amount of bandwidth needed to recirculate packets highly depends on the input rate. When the input rate is high, only a few packets need to recirculate. When the input rate is low, more packets need to recirculate (so as to build up a queue). Consequently, the amount of bandwidth that must be allocated to recirculating packets may vary over time in response to the input rate. Unfortunately, today's switches do not allow to dynamically modify the role of a port, *i.e.,* moving it from a loopback port to a normal outgoing port. An operator must statically configure how many ports will be used for recirculating packets. In the worst case, an operator must use half of the switch bandwidth to recirculating each incoming packet. This constraint makes the multicast-based approach more efficient, as it does not suffer from this problem.

We leave challenges related to packet drops as future work.

IV. PRELIMINARY EVALUATION

We show a preliminary evaluation of TURBOSWITCH based on a simplified switch simulator (implemented in Python) and an ASIC-based prototype. The goal is only to demonstrate the expected behaviour of the system and not to show a fully working solution modeling all the possible implementation details. We use the simulator to reason about the queue occupancy, its memory requirements, the recirculation overheads, the overall packet processing latency, and the output throughput.

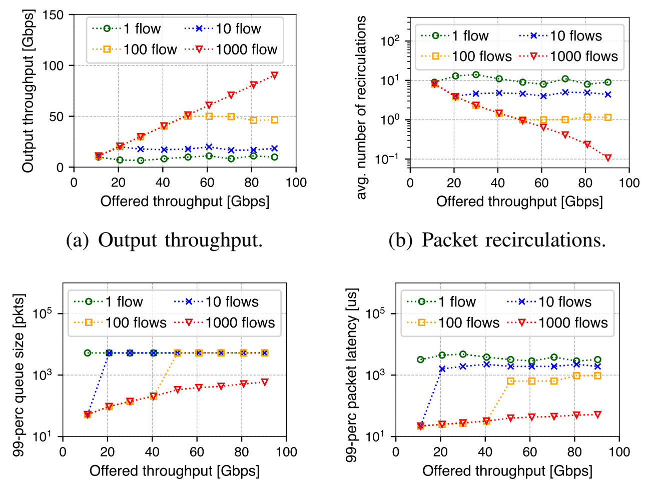
Switch model. We model a switch with three 100-Gbps ports: an input, an output, and a loopback port. Note that only the recirculation-based approach uses the loopback port. Our simulator processes incoming packets arriving according



(a) Output throughput (b) Packet recirculations.

(c) Queue size. (d) Packet processing latency.

Fig. 4. TURBOSWITCH with a recirculation-based approach <u>without</u> packet ordering guarantees.



(a) Output throughput. (b) Packet recirculations.

(c) Queue size. (d) Packet processing latency.

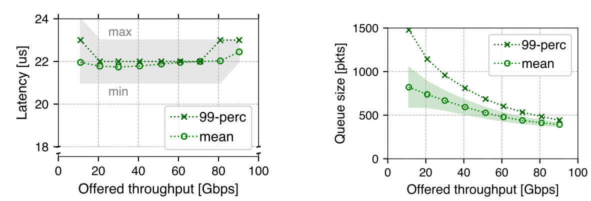
Fig. 5. TURBOSWITCH with a recirculation-based approach <u>with</u> packet ordering guarantees.

to a Poisson process with a given input throughput. We set the minimum packet size to 64 B and the maximum packet size to 1500 B. The packet size is generated according to an exponential distribution with an average value of 1000 B. We set the maximum queue size to 850 kB, beyond which the switch drops incoming packets. In the recirculation approach, a packet is sent out after it spent at least 20 μs in the system. In the multicast approach, we add a 1 μs delay for updating the queue size information.

Recirculating packets self-clocks the system to the correct queue size. We generate an input rate from 10 to 90 Gbps. Fig. 4 shows the different metrics achieved during the simulations. On all graphs, we show the average and 99-th percentile values as well as the standard deviation (shown as a green area). Fig. 4(a) shows that TURBOSWITCH is able to forward traffic at any given input rate. Fig. 4(b) shows that the number of per-packet recirculations decreases when increasing the input rate. It is important to note that the recirculation port is always fully utilized in order to build a queue and keep it stable. Fig. 4(c) shows the queue size (in packets), which grows when increasing the input rate. This is in line with Fact 2. Also, we observe (not shown in the graph), that the maximum queue size is 450 kB (at an input rate of 90 Gbps), which is in line with Theorem 1 indicating a 500 kB upper bound. Fig. 4(d) shows the packet processing latency, which is above the 20 μs of the NF processing latency and is below 60 μs according to Theorem 1.

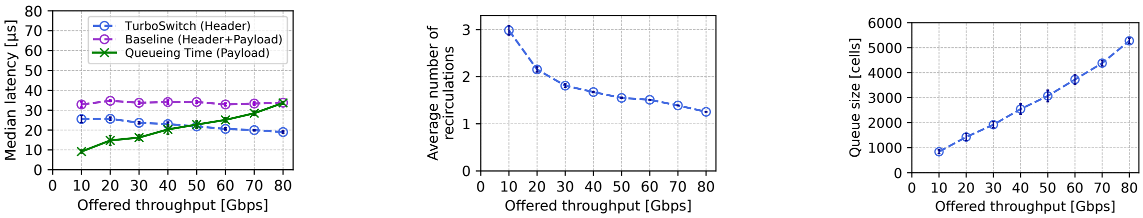
Ensuring packet ordering when recirculating packets reduces throughput when few flows exist. We now investigate in Fig. 5 the impact of guaranteeing packet ordering with the recirculation-based approach. In this preliminary evaluation, we generate packets belonging to 1, 10, 100, or 1000 distinct flows using a round-robin approach, and we assume packet drops can immediately be detected so that the switch can move to the next packet. We only show the 99-th percentile results and the average result for the number of recirculations. Fig. 5(a) shows that even with just only 1000 distinct flows the system copes with the maximum throughput of 90 Gbps. In fact, we observe in Fig. 5(c) that the queue is completely full for any input rate above 60 Gbps for all the other cases with 1, 10, or 100 distinct flows. In these cases, the switch must recirculate packets more times to guarantee packet ordering (as shown in Fig. 5(b)), causing the queue to grow excessively. Fig. 5(d) shows the packet latency, which is below 60 μs with 1000 flows.

The multicast-based approach correctly sizes the queue to respect the NF latency. We now evaluate in Fig. 6 the simplified multicast-based approach described in §III-B with a 1 μs delay to fetch the queue occupancy. We first note (not shown in the graphs) that the output throughput of TURBOSWITCH always matches the input rate without dropping any packet. Fig. 6(a) shows the average and 99-th percentile packet processing latency as well as the minimum and maximum latencies (shown as a gray area). We note that the minimum latency is always above the NF packet processing threshold of 20 μs and the latency is very stable. Fig. 6(b) shows the queue occupancy in packets. We observe something counter-intuitive, *i.e.*, at lower rates, the number of packets in the queue is higher than at higher rates. Yet, when we look at the queue occupancy in bytes (not shown in



(a) Packet latency. (b) Queue size.

Fig. 6. TURBOSWITCH (multicast-based approach).



(a) Packet processing latency.
(b) Packet recirculations.
(c) Average queue size.

Fig. 7. Recirculation-based TURBOSWITCH on an ASIC implementation.

the figure), we observe that it grows with an increasing input rate, *i.e.,* from 300 kB to 400 kB. This phenomenon arises because at lower rates the system creates more copies of a packet than at higher rates. When the multicasted packet is a small packet, TURBOSWITCH creates a very large amount of copies of this packet, thus enqueuing more packets than at higher rates, where the packet size distribution in the queue is more similar to the packet size distribution of the input rate (*i.e.,* centered around 1 kB packets). This explain the above counter-intuitive behaviour. Finally, our simulations show that, for small packets, the system must create a large number of multicast copies of the packet.

Would TURBOSWITCH work for real? We implement a proof-of-concept of the recirculation-based approach on the Intel Tofino programmable ASIC, incorporating specific adaptations from the design of §III-A. Since it is not possible to change the output port after dequeing a packet, we store processed headers before the queues on the switch, so that reconstructed packets can directly choose the correct output port (without an additional recirculation). We connect the switch to a server which acts as traffic generator and another server which runs a forwarder NF with an average latency of 25μs. Both servers are connected with 100-Gbps links. Tests are repeated 10 times. Fig. 7(a) shows three lines: *i)* the median NF-to-switch latency using TURBOSWITCH (blue line) where the NF only process the packet header, *ii)* the median NF-to-switch latency using a baseline (purple line) that forwards 1024 B packets without splitting the packet, and *iii)* the queuing time in TURBOSWITCH (green line). On the x-axis, we increase the input rate. The packet processing time on the NF benefits from processing only the headers (blue line compared to purple line) as already demonstrated in recent work [2], [3], reducing the latency from 30 to 20μs. Also, we note that by increasing the throughput, the queuing time of the payloads increases linearly with a value that keeps below 40 μs, which confirms Theorem 1. The linear increase is explained by the average number of payload recirculations shown in Fig. 7(b). At low loads, queues are smaller as packets recirculate more often, confirming the results shown in Fig. 4(b). The low number of recirculations (*i.e.*, 3 instead of 9) seems to come from the packet scheduler of the ASIC switch. Fig. 7(c) shows the average queue size with respect to the injected throughput. The queue size is in number of cells (each cell is a 80-bytes block of the pipe buffer pool [14]). As expected, the queue size grows when increasing the input rate, hence the results of the Tofino implementation are in line with the findings of § IV and, again, confirm Fact 2. Moreover, converting the number of cells in bytes, we have that the queue size is 422 kB at 80 Gbps, which is lower than 500 kB by Theorem 1.

V. CONCLUSION

We presented TURBOSWITCH, a novel approach for supporting high-speed NF deployments. We exploit existing queue-based memory buffers on ASIC switches to store packet payloads while their headers are processed on external NFs. In this way, TURBOSWITCH avoids wasting bandwidth on the switch to store/retrieve payloads. Our prototype shows the potential for achieving full line-rate multi-Tbps advanced NF processing on a single switch and external NF processor.

ACKNOWLEDGEMENTS

This work has been partially supported by Vinnova (the Sweden's Innovation Agency), the Swedish Research Council (agreement No. 2021-04212), and KTH Digital Futures.

REFERENCES

[1] P. Bosshart, G. Gibb *et al.*, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 99–110, 2013.
[2] M. Scazzariello, T. Caiazzi *et al.*, "A High-Speed stateful packet processing approach for tbps programmable switches," in *NSDI*, 2023.
[3] B. Pismenny, L. Liss *et al.*, "The Benefits of General-Purpose On-NIC Memory," in *ASPLOS*, 2022.
[4] S. Goswami, N. Kodirov *et al.*, "Parking packet payload with p4," in *CoNEXT*, 2020.
[5] D. Kim, Z. Liu *et al.*, "TEA: Enabling State-Intensive Network Functions on Programmable Switches," in *SIGCOMM*, 2020.
[6] C. Zeng, L. Luo *et al.*, "Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing," in *NSDI*, 2022.
[7] H. Ghasemirahni, T. Barbette *et al.*, "Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets," in *NSDI*, 2022.
[8] T. Lévai, F. Németh *et al.*, "Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives," in *NSDI*, 2020.
[9] M. A. Johnson and M. H. Moradi, *PID control*. Springer, 2005.
[10] D. Dutt, *BGP in the Data Center*. O'Reilly Media, 2017, page 11.
[11] J. Zhou, M. Tewari *et al.*, "Wcmp: Weighted cost multipathing for improved fairness in data centers," in *EuroSys*, 2014.
[12] E. Vanini, R. Pan *et al.*, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *NSDI*, 2017.
[13] M. Handley, C. Raiciu *et al.*, "Re-architecting datacenter networks and stacks for low latency and high performance," in *SIGCOMM*, 2017.
[14] Intel, "P4$_{16}$ Intel Tofino Native Architecture," 2021, https://raw.githubusercontent.com/barefootnetworks/Open-Tofino/master/PUBLIC_Tofino-Native-Arch.pdf.