Adaptive Batching for Fast Packet Processing in Software Routers using Machine Learning

Peter Okelmann*, Leonardo Linguaglossa†, Fabien Geyer*, Paul Emmerich*, Georg Carle*
*Technical University of Munich †Telecom Paris
okelmann@in.tum.de, {fgeyer,emmericp,carle}@net.in.tum.de linguaglossa@telecom-paristech.fr

***Abstract*—Processing packets in batches is a common technique in high-speed software routers to improve routing efficiency and increase throughput. With the growing popularity of novel paradigms such as Network Function Virtualization, advocating for the replacement of hardware-based networking modules towards software-based network functions deployed on commodity servers, we observe that batching techniques have been successfully implemented to reduce the HW/SW performance gap. As batch creation and management is at the very core of high-speed packet processors, it provides a significant impact to the overall packet processing capabilities of the system, affecting latency, throughput, CPU utilization and power consumption. It is commonly accepted to adopt a fixed maximum batching size (usually in the range between 32 and 512) to optimize for the worst case scenario (i.e. minimum-size packets at full bandwidth capacity). Such approach may result in a loss of efficiency despite a 100% utilization of the CPU. In this work we explore the possibilities of enhancing the runtime batch creation in VPP, a popular software router based on the Intel DPDK framework. Instead of relying on the automatic batch creation, we apply machine learning techniques to optimize the batching size for lower CPU-time and higher power efficiency in average scenarios, while maintaining its high performance in the worst case.**

I. Introduction

Software packet processing has become a commonly adopted alternative to costly, expensive hardware-based processing engines [1], [2]. Together with the inherent flexibility advantages of software-based solutions w.r.t. hardware counterparts, a current trend shows that the performance gap is also diminishing [3]. As a consequence, several libraries for high-speed packet processing on pure software such as the Intel's DPDK[1] or Netmap [4] are being used as building blocks for a flourishing ecosystem of software middleboxes capable of performing multi-10-Gbps packet processing on a single CPU core of commercial off-the-shelf (COTS) servers.

Modern tools for software packet processing, also known as *software routers*, incorporate many optimizations such as processing packets in batches and adopting a kernel-bypass approach to access the Network Interface Cards (NICs) with pure user-space drivers and minimize the interference of low-level system calls by the operating system. In particular, batching is usually adopted in conjunction with a busy polling behavior: the CPU continuously performs a loop to verify if any packet is received at the NIC, then it uses a minimalistic batch creation algorithm to process a full batch of packets (as opposed to per-packet processing) and it repeats the loop at the end of the processing. Batching and busy polling are very effective in high-load scenarios, where the cost of interrupt handling per packet could saturate the CPU. In particular, it has been shown that increasing the batch size positively correlates with a significant improvement in the packet processing rate, up to a certain saturation threshold [5], [1]. Therefore, the achievable throughput is maximized at the cost of a 100% CPU utilization even in the case of low-load scenarios, resulting in a lot of wasted CPU cycles and high power consumption.

While the maximum batch size is fixed (usually 32 to 512), the actual size depends on the number of packets waiting in the NIC's input queues. But the actual batch size also affects the processing efficiency, with small batches requiring more clock cycles per packet than large ones. This causes a feedback loop, where oscillating batch size can be observed in scenarios where the input load does not fully saturate the CPU. Such a batching approach provides opportunities for improvement: ideally, in a non-saturated regime (i.e., no packet loss) the CPU can be relieved of some processing if we keep the batch sizes large enough to maintain the processing efficiency.

In this paper[2], we propose an algorithm to dynamically allocate batch sizes depending on the traffic condition instead of the classical busy polling approaches. With the help of a large dataset collected over hours of experiments with a real packet processing engine, we first develop machine learning techniques to find the optimal batching size for different load scenarios. We then deploy our training model within a software router, and assess the impact of our approach in terms of saved clock cycles. The remainder of the manuscript is organized as follows: Section II provides a background on the relevant architectural aspects of the software router of our choice (namely, VPP [6]) and the related work. In Section III we analyze the possible improvements for the batching algorithm and present our design space. We then evaluate our system in Section IV and Section V concludes the manuscript.

II. Background and related work

In a softwarized network scenario, a *software router* is a piece of code running on a general-purpose server which is responsible for moving packets from one NIC (or more) to a network application for further processing. Since NICs can be both physical or virtual, software routers are fundamental components of virtualized network systems. When a NIC is

[1] `https://www.dpdk.org/`
[2] Our source code and scripts: https://github.com/pogobanane/vpp-testing

978-1-6654-0522-5/21/$31.00 ©2021 IEEE

equipped with multiple hardware queues, a software router process is usually bound to a single queue and executed by a single CPU, to allow horizontal scalability while, at the same time, avoiding inter-process interruptions [7]. The majority of high-speed software routers can be executed within Linux environments, and make extensive usage of low-level libraries such as Intel's DPDK or netmap [4] rather than relying on the standard libraries for packet processing. Such libraries are optimized to maximize the computational efficiency of the packet processing application, and provide the additional advantage of avoiding the overhead of the Linux kernel [1].

A. *Vector packet processing*

As a use case for our work, we select Vector Packet Processing (VPP), a high speed packet processor originally developed by Cisco and recently released as an open-source Linux Foundation's project named FD.io [6]. VPP provides a rich feature set for a wide range of hardware and architectures, and adopts most of the popular design choices to improve the packet processing rate [1]. To improve modularity and ease of programming, most of VPP's features are developed as individual plugins, that are further organized as nodes in a *processing graph*, which represents the desired packet processing applications. When a physical NIC is controlled by a DPDK driver, the first node accessed upon packet reception is the `dpdk-input` node, which is responsible for querying packets from the NIC via the DPDK library, creating a batch with the received packets and passing the full batch to the next node. Subsequent nodes can differ, depending on the required network stack to be accessed. For example, in the case of IPv4 packets, a following node may be the `ip4-input` that will parse the IPv4 headers, or a `ip6-input` that will deal with IPv6 packets. At the end of the processing, a final output decision is taken at the `dpdk-output` node which can choose to forward packets to another physical or virtual NIC.

With the adoption of receive-side scaling (RSS), VPP's main thread can distribute the incoming traffic to multiple worker threads. Each thread then runs its own instance of the processing graph. Worker threads can be conveniently pinned to cores and assigned a scheduler via VPP's configuration. This allows reaching higher throughput when processing multiple traffic flows. Since we are interested in changing the batch creation behavior, we focus our investigation on the `dpdk-input` node.

B. *Batch creation and CPU behavior*

The unmodified version of `dpdk-input` implements a busy-loop which continuously polls new packets from the NIC using the call `rte_eth_rx_burst`. If no packets have arrived, it simply returns to the beginning of the loop, which leads to the same node being called again. In this way, while the main thread is busy waiting for new packets, the CPU is continuously utilized at 100%. If a poll detects some packets queued at the NIC, the DPDK library tries to retrieve as many packets as possible until the maximum batch size has been reached (defaults to a value of 256). It is worth noting that when the DPDK node detects less than 32 packets waiting at the NIC,

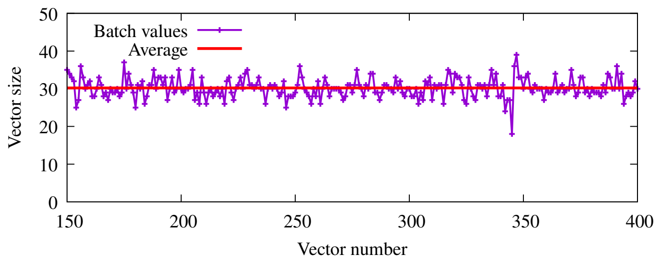

Figure 1: Oscillation of batch sizes for a L2 forwarding function that processes a constant 5 Gbit/s traffic.

the loop returns immediately as the driver assumes that no more packets will be arriving at the queue. This proves useful in low-load scenarios, as packets are immediately batched and submitted into the processing graph to keep latency low.

The size of soon-to-be-processed batches highly affects the computational efficiency of the underlying CPU. Considering a constant-bit-rate (CBR) scenario, when a large batch is received the CPU efficiency is very high [1], and as a result, the processing time per packet is low. Since the polling loop can quickly return to retrieve more packets, the next poll will retrieve less packets because of the constant bit rate. A smaller batch, will result in a lower efficiency and, therefore, a higher processing time per packet. This will in turn result in more packets being queued at the NIC, and another new poll with a larger batch size. This oscillating behavior can be observed in Figure 1, which shows the number of packets in a batch as a function of the time, for an average load of 5 Gbit/s. This behavior leads the CPU to switch between higher power-consumption condition, back to low-energy consumption. Moreover, batch sizes also have an impact on latencies of the packets [8]. Keeping in mind that the CPU is continuously utilized at 100 %, we propose a different approach which tries to (i) minimize the oscillating behavior, (ii) keep the CPU efficiency always at its maximum and (iii) release the CPU occupancy by using an idle state which will free some clock cycles (that can be used by other concurrent applications).

C. *Related work*

Optimizations of batching behavior are closely related to our work: for example, SmartBatching [8] aims at adapting the batching behavior according to an analytical model derived from the input load, which improves both CPU behavior and latency. Similarly, Metronome [9] is an approach to replace the continuous polling with a sleep and wake intermittent mode and an optimized CPU `sleep` function. Analytical modeling is becoming widely adopted to provide previsions on key performance metrics such as the packet loss or the expected batch sizes, as done in [10]. Our work differs from the previous in that it relies on machine learning to adapt to the incoming input rate, rather than on analytical modeling. This way, we can relax the assumptions on the input traffic pattern, as we just need to train our model with realistic traffic. We rely on the standard Linux `nanosleep` without any additional kernel module. A different approach is adopted by Shenango [11],

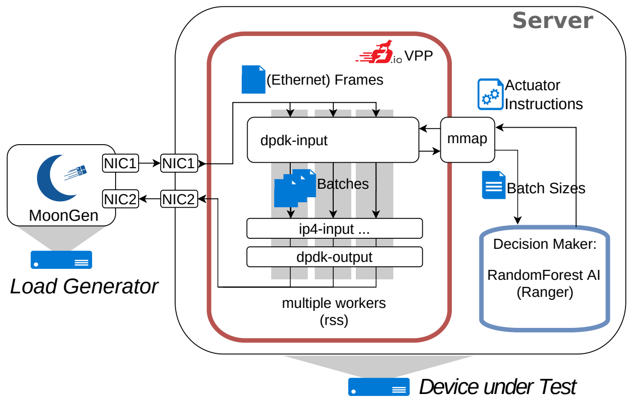

Figure 2: The testbed used for our experimental evaluation, which includes the MoonGen traffic generator, the device under test and the AI component.

which divides the available CPU cycles into fine-grain slots, while a separate orchestrator reallocates the CPU cores and steers the input traffic depending on latency requirements. Our method is more flexible in that we do not require a fixed time-slot for the reallocation of CPU cycles.

III. Design space

As explained in Section II, the batch size used by VPP depends on the load at the NIC. Although VPP tries to create maximum-size batches in high-load, it may be beneficial to find a different value in lower load scenarios. For example it can be useful to keep batches artificially large while, at the same time, saving some CPU clock cycles for further processing. In the scenario depicted in Figure 1 this may help to reduce the batch size oscillations as well as freeing some CPU clock cycles which can be used for different processing. In a virtualized environment, CPU cores may be shared with other applications. With a tuned value for the batch sizes, the `dpdk-input` node could then free the CPU core for other applications. This can be done with the UNIX `nanosleep` syscall. Another simple but effective measure could be to scale the frequency of the CPU clock in order to reduce the power consumption and reduce the clock cycles allocated for the busy-polling. Finally, it can be possible to enable the interrupt mode for very low load scenarios, though this approach would require a fine tuning of the switching threshold. Table I summarizes the possible actions. We now show how to estimate the optimal values for the batch sizes.

A. Decision making via ML

The aforementioned modifications are beneficial only in certain scenarios. Their usage must be adapted to the current load situation by a *decision maker* (DM). This process can be defined through modeling, which however is prone to errors as it is hard to create a representation of the system that is sufficiently detailed to capture the low-level details that are essential for our processing [10]. As the actions influence latency, throughput, CPU usage and power consumption, the decision making process must optimize for all of those. Additionally, most actions will also affect the subsequent state of the system, thus incurring in feedback actions and non-linear effects. We opt for a machine learning solution as an alternative to classical analytical approaches, as it can be used to approximate a solution for such a complex problem, without the necessity of manually modeling and tuning the system model. The DM uses a list of the last batch sizes as input for its actions. In combination with the parameters to optimize for and the possible actions to tune, this results in a problem of high dimensionality. Our ML decision maker is used to regularly update the threashold configuration for using actions.

B. Architecture overview

We now describe the proposed architecture, as shown in Figure 2. The device under test (DUT) consists of two parallel components: the software router, and the decision maker. Every time the `dpdk-input` node submits a batch of packets to the processing graph, it also communicates the batch size to the DM. The ML algorithm then runs its predictions and returns the new, updated action instructions which are in turn read by VPP. For the IPC communication we adopt non-blocking I/O in order to keep high throughput performance.

For the DM component, it is essential to use fast ML techniques, as otherwise the efficiency advantages would be negated by the resource hungry machine learning component. We selected random forests and ranger [12], as they are efficient and easy to integrate in VPP.

In theory, it is possible to alter the state of the system by adopting a combination of all the actions shown in Table I, depending on the severity of the impact on the processing. For example, switching to interrupt mode would reduce the load on the CPU, at the cost of a severe performance degradation. However, the base Ranger version comes with a limited interface with no support for multi-dimensional variables. Therefore the batch selection must be controlled by a single variable (and thus, a single action can be used). Although all the mechanisms shown in Table I are implemented, we focus here on `nanosleep` actions controlled by a single integer.

Description	Implemented	Used
release the CPU (`nanosleep`)	✓	✓
delay the polling (`rte_delay_us`)	✓	✗
interrupt mode (~~`rte_eth_dev_rx_intr_*`~~)	✓*	✗
CPU freq hints (`rte_power_freq_up`)	✗	✗

Table I: Proposed actions and their usage by ranger.
* : Implemented, but not functional.

C. Random Forest Training

The ranger API is used for training a forest taking the latest used batch sizes of VPP as input, and to predict the best time to nanosleep for. As presented in Figure 2, VPP and ranger run on the DuT, while load scenarios are performed by the load generator running MoonGen [13]. White box measurements like clock cycle counting are conducted on the DuT, and black box measurements like latency and throughput are collected by the load generator.

Method	msg/s	avg (µs)	min (µs)	max (µs)	std dev (µs)
mmap	867,092	1.103	1.024	5.376	0.178
shm	726,068	1.377	1.320	1.024	0.334
fifo	76,029	13.037	10.496	27.396	0.751
pipe	59,972	16.674	14.557	120.320	3.960

Table II: Comparison of IPC with 1000 messages of 4096 bytes each

In order to train the random forest, an iterative process is used. After each run of the performance measurements, the success of the forest is evaluated using a reward function. The reward is then used to refine the prediction values to train for which combined with the newly collected ranger input batch sizes make up the new training set. Finally, the next iteration of the forest can be trained and the next training iteration begins.

We limited our evaluation to six scenarios with different packet rates: 2, 10, 500, 1000, 5000 and 7500 Mbit/s. Since we focus on the `nanosleep` action, we selected low load scenarios where freeing CPU cores is beneficial, but also included larger loads to avoid over-fitting.

In each training iteration those six scenarios are run and used to refine the training set. Finding the correct prediction results relies on a reward function r. It weights the average latency l in µs, the average throughput t in packets per second and the CPU cycles c used by VPP as follows:

$$r(l,t,c) = -0.5l + \frac{4000t}{c * 10^{-7}}$$

Afterwards the range of the reward is limited to the range $[0,-1]$ using the expected minimum and maximum reward values $r_{min} = -10$ and $r_{max} = 120$:

$$p_{deviate}(r) = 1 - \frac{r}{|r_{min} - r_{max}|}$$

This is the probability with which the latest prediction p_{last} should be changed. Next the new prediction result to be trained for is drawn using a random function with a normal distribution, p_{last} as center and s as standard deviation. It is based on p_{last}, $p_{deviate}$ and a constant to guarantee the continued exploration of new values $c_{explore} = 5$:

$$s(p_{deviate}) = 100 * p_{deviate}^5 + c_{explore}$$

The more the reward r rises, the smaller becomes $p_{deviate}$ which in turn results in an aggressive reduction of s. Using that system over many training iterations, the random forest output is able to converge.

IV. Numerical evaluation

We numerically evaluate in this section the different components of our architecture. We also illustrate the impact that our machine-learning based software router has on the overall performance. Our benchmarks are performed on a server with an Intel Xeon CPU E31230 at 3.20 GHz.

The traffic generated by MoonGen consists of 64 B packets since it is most demanding scenario for the software component of software routers. Other parameters which were not tested, like packet size, can have an impact though as well on packet processing times and thereby also influence latency.

Stage of Integration	Throughput	Ratio
Unmodified VPP	14.15 Mpps	100 %
Logging only	13.95 Mpps	99 %
Logging + Exporting	13.94 Mpps	99 %
...+ Exporting + Ranger load	11.57 Mpps	82 %
...+ Final trained forest	12.26 Mpps	87 %

Table III: Maximum throughput at different stages integration.

A. Inter-Process Communication

We evaluate here the solution used for communicating batch sizes and instructions between VPP and ranger. VPP already has the `elog` system[3] for in place logging. While small logging events should be performant, previous works [14] show that it can significantly impact the performance. Using an open-source benchmarking suite[4], we evaluated alternative IPC channels (see Table II). Based on those results, we built a more efficient communication channel using mmap.

Table III summarizes the impact of data collection and export on VPP's throughput. Running VPP while logging all batch sizes into the shared memory, results in a maximum throughput of 99 %. Running ranger for a single prediction, meaning exporting the ringbuffer only once to ranger, results in a similar throughput performance of 99 %. When running the ringbuffer export and prediction in an endless loop, VPP's throughput drops to 11.57 Mpps which corresponds to 82 % of the performance of unmodified VPP. Compared to other approaches using `elog` [14], showing a performance loss of 2 to 3 times, our mmap-based IPC proved to be more efficient.

Scenario	Pred./s	Std. dev.	Batches/Pred.
Random data	22 106	168	2.6
Real IPC	712	46	82.3

Table IV: Ranger predictions rate

B. Ranger performance

We evaluate the prediction rate of ranger with the help of Table IV, which illustrates how many batches may be processed on a fully utilized 10 Gbit/s link per prediction. We first benchmark ranger with random data and our results showed 22 106 predictions/s in average. With real data on the IPC channel, the performance drops to only 712 predictions/s. The number of packet batches n_v processed between two ranger predictions can be calculated from the packet throughput t_r, the average size of a batch v_s and the prediction rate p_r: $n_v = t_r/(v_s * p_r)$

With $p_r = 22\,106$, we obtain 3 batches per prediction which is close to optimal. With $p_r = 712$, each prediction will be used for the next 82 batches. This performance gap could be improved with an optimized mechanism for the interaction between the AI and the DUT component.

[3]`https://wiki.fd.io/view/VPP/elog`
[4]`https://github.com/goldsborough/ipc-bench`

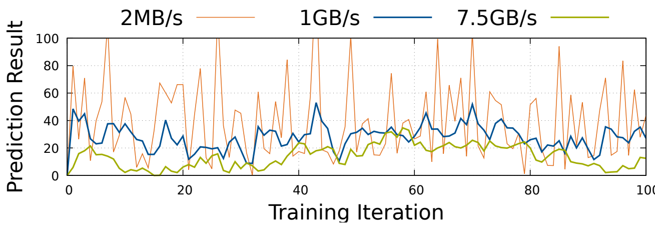

Figure 3: Convergence of the training in three scenarios.

C. Training

For each load scenario presented in Section III-C, we evaluated the prediction over the training iterations of the forest. Figure 3 shows that a convergence cannot be observed. For the higher loads of 1 Gbit/s and 7.5 Gbit/s, a trend emerges though: the random forest predicts higher nanosleep times for smaller load scenarios which is what a human expert would expect. For the smallest traffic rate used, the training could not converge. A reason could be that satisfying the reward function in situations with that little traffic is really hard because the CPU cycles per packet used by VPP seem to rise non-linearly for those scenarios. Our experiments show that the nanosleep time has to be several orders of magnitude higher than 50 µs to get to a CPU cycle efficiency expected by the reward function.

D. Validation and comparison of the system

Finally, we evaluate our architecture by measuring the CPU utilization in different scenarios and comparing it to an unmodified VPP (see Figure 4). Unmodified VPP has the worker thread running on CPU1. Its utilization is 100 % regardless of the offered load. The second core runs nothing and therefore has a utilization of 0 %. Depending on the traffic its (avg, max) latencies are between (5, 6)µs and (12, 65)µs.

Modified VPP with ranger updating the optimization instructions on CPU2 constantly utilizes about 98 %. The worker thread of VPP on CPU1 now shows a different behavior. When offered no load, the nanosleep time is set to 30 by the forest. This results in a utilization of only around 20 % on CPU1. When offered more load (1000 Mbit/s around time 20 s in Figure 4), the nanosleep time drops and CPU1 raises to 45 % load to process the packets. From offering 1200 Mbit/s of load onward (at time 31 s in Figure 4), VPP's worker thread starts hitting the upper limit of available cycles of CPU1. The latencies range from (12.5, 54)µs to (13.4, 101)µs. At a throughput of 12.26 Mpps the maximum performance of the system is finally found.

V. Conclusion

We showed that the CPU utilization of VPP could be reduced in low load scenarios using random forests for finding optimization parameters at runtime. Regardless of the added context switches and complexity, the throughput performance in high load situations is reduced by only 13 %. Although a separate core is fully utilized by the ranger thread, future work could use the same core to save CPU time off VPP worker threads on multiple cores. Among possible optimizations, dynamically reducing ranger's refresh rate may be a promising one. Further research is also required for comparative study and evaluation of the effects on both throughput and latencies combined, because of the trade-off between CPU efficiency and processing times. Finally, exploring other random forest implementations may open possibilities for using more actions, improving training and implementing online learning to react to new, unknown traffic patterns.

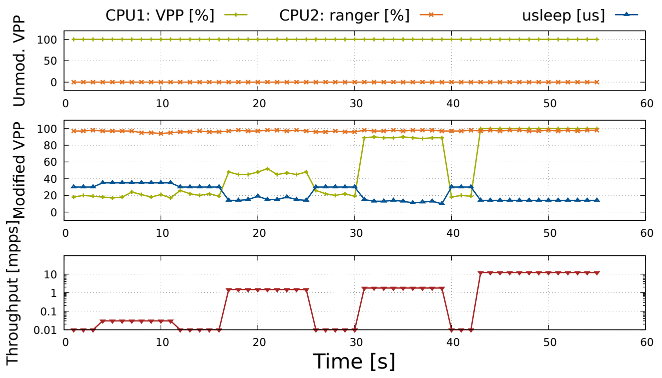

Figure 4: A comparison under the same load scenario. Unmodified: CPU1: VPP worker, CPU2: nothing. Modified: CPU1: VPP worker, CPU2: ranger.

References

[1] L. Linguaglossa, D. Rossi, S. Pontarelli, D. Barach, D. Marjon, and P. Pfister, "High-speed data plane and network functions virtualization by vectorizing packet processing," *Computer Networks*, 2019.

[2] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *Proc. of ACM/IEEE ANCS*, 2015.

[3] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi, "Survey of performance acceleration techniques for network function virtualization," *Proc. of the IEEE*, 2019.

[4] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *USENIX Security 12*, 2012.

[5] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proc. of ACM/IEEE ANCS*, 2015.

[6] "What is VPP?" https://wiki.fd.io/view/VPP/What_is_VPP%3F, accessed on 2020-12-01.

[7] T. Herbert and W. de Bruijn, "Scaling in the linux networking stack," https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2011.

[8] M. Miao, W. Cheng, F. Ren, and J. Xie, "Smart batching: A load-sensitive self-tuning packet I/O using dynamic batch sizing," in *Proc of HPCC/SmartCity/DSS*.

[9] M. Faltelli, G. Belocchi, F. Quaglia, S. Pontarelli, and G. Bianchi, "Metronome: Adaptive and precise intermittent packet retrieval in DPDK," 2020.

[10] S. Lange, L. Linguaglossa, S. Geissler, D. Rossi, and T. Zinner, "Discrete-time modeling of NFV accelerators that exploit batched processing," in *IEEE INFOCOM 2019*, 2019.

[11] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *USENIX NSDI*, 2019.

[12] M. N. Wright and A. Ziegler, "ranger: A fast implementation of random forests for high dimensional data in C++ and R," *arXiv:1508.04409*, 2015.

[13] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A scriptable high-speed packet generator," in *IMC*, 2015.

[14] L. Linguaglossa, F. Geyer, W. Shao, F. Brockners, and G. Carle, "Demonstrating the cost of collecting in-network measurements for high-speed VNFs," in *Proc. of TMA*, 2019.