Providing Telco-oriented Network Services with eBPF: the Case for a 5G Mobile Gateway

Federico Parola, Fulvio Risso
Department of Computer and Control Engineering
Politecnico di Torino, Italy
name.surname@polito.it

Sebastiano Miano
Queen Mary University of London, London, UK
s.miano@qmul.ac.uk

2021 IEEE 7th International Conference on Network Softwarization (NetSoft) | 978-1-6654-0522-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/NetSoft51509.2021.9492571

***Abstract*—Although several technologies exist for high-speed data plane processing, such as DPDK, the above technologies require a rigid partitioning of the resources of the system, such as dedicated CPU cores and network interfaces. Unfortunately, this is not always possible when running at the edge of the network, in which a few servers are available in each cluster and a mixture of data and control plane services must coexist on the same hardware. In this respect, eBPF can become a better alternative thanks to its integration in the vanilla Linux kernel, which enables contemporary support for data and control plane services, hence enabling a more efficient u sage o f the (scarce) computing resources. This paper proposes the first proof-of-concept open-source implementation of a 5G Mobile Gateway based on eBPF/XDP, highlighting the possible challenges (e.g., to create traffic p olicers, a s b uffering i s n ot a vailable in eBPF) and the resulting architecture. The result is characterized in terms of performance and scalability and compared with alternative technologies, showing that it outperforms other in-kernel solutions (e.g., Open vSwitch) and is comparable with DPDK-based platforms.**

***Index Terms*—Network Functions, eBPF, XDP, 5G Mobile Gateway, 5G User Plane Function.**

I. Introduction

With the diffusion of Multi-access Edge Computing (MEC), the 5G Mobile Gateway, implementing the User Plane Function (UPF), is increasingly deployed nearby the Radio Access Network (RAN), enabling telcos to provide services at close proximity to mobile users.

In this scenario, high performance data plane technologies such as DPDK may not be appropriate because of their complexity, with a support model that requires significant investment to maintain and integrate due to proprietary drivers. Furthermore, they take full control on portions of the server, relying on polling to retrieve packets (hence requiring dedicated CPU cores) and using their own drivers to control the NIC, which cannot be longer used by the operating system TCP/IP stack. This behaviour creates a rigid partitioning of the resources of the system, a constraint that is not acceptable in "mini" data centers, which are often deployed at the edge of the network (i.e., close to each 5G site), where resources should be dynamically shared between both data and control plane services. In this scenario, eBPF/XDP can represent a better solution; while its raw performance are inferior to DPDK-based platforms [1], its better integration with vanilla Linux kernel makes it suitable to be used with different kind of workloads, and transparently be integrated with cloud orchestrators such as Kubernetes.

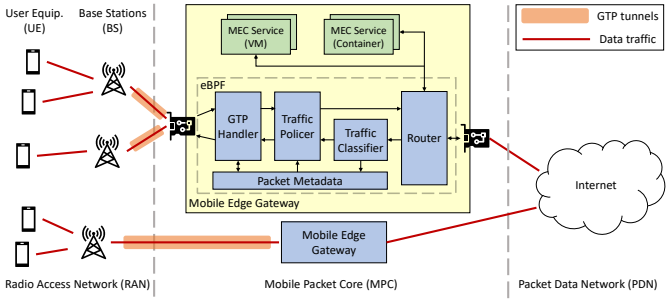

Figure 1: Mobile Gateway prototype architecture.

However, no eBPF/XDP implementations of a MGW exist so far, with the closest available ones being proof-of-concept implementation relying on different frameworks for software network functions and software switches (e.g. Open vSwitch) presented in [2]. Other works like [3] and [4] focus on improving the performance of the Gateway (in the first case proposing its offload to programmable switch ASICs) but do not consider integration issues discussed in this work. The lack of an eBPF Gateway is due to the event-driven nature and limitations of the eBPF platform, which poses non trivial challenges in the implementation of key components such as shapers/policers, and the difficulties in writing complex data plane services. This paper aims at filling this gap, presenting the first proof-of-concept open-source[1] implementation of a mobile gateway and its preliminary benchmarking. This work confirms the feasibility of a MGW in eBPF/XDP and shows that the performance of this first PoC implementation greatly outperform other in-kernel solutions and it is comparable with more efficient DPDK-based platforms.

II. Design

Figure 1 illustrates the high-level architecture of a mobile network, with different instances of a mobile gateway that are placed on the the same servers where the others MEC services are running. The Mobile Gateway handles the data traffic of the user equipment (UE), encapsulated into GTP-u tunnels and delivered to the 5G Mobile Packet Core through radio Base

[1] https://github.com/polycube-network/polycube/

Stations (BSs). It replaces and merges the roles carried out by the data plane of the Serving Gateway and PDN Gateway in the LTE Evolved Packet Core. Its functionalities include routing and forwarding of traffic between the Access Network and an external Packet Data Network (e.g. the Internet), management of GTP-u tunnels, access control, per-flow QoS, guaranteed and maximum bit rate, traffic charging, traffic monitoring and the support of user mobility across different radio Base Stations.

A. Overall Architecture

We selected some of the most significant functionalities and implemented them as four separate in-kernel network services, leveraging one or more eBPF programs deployed through the Polycube [5] eBPF framework. This framework provides useful abstractions for the creation of eBPF-based network functions and their chaining to compose complex services. Every module is composed by (*i*) a user space control plane, accessible through a RESTful API, and (*ii*) an in-kernel data plane, leveraging one or more eBPF programs.

A packet flowing in the uplink direction (from the access to the data network) crosses a *GTP Handler* in charge of removing the GTP encapsulation, a *Traffic Policer*, that applies rate limit to flows in order to enforce QoS, and a *Router* that forwards the traffic to either the external network or towards a service running on the local server. In the opposite direction, the packet is received by the router, processed by a *Classifier* that determines the GTP tunnel and QoS flow it belongs to, handled by the *Policer* and eventually encapsulated in GTP.

To simplify the implementation of the prototype while still being able to compare with other solutions (sec. III), we adopted a QoS model with one class associated to each GTP tunnel (identified by a Tunnel Endpoint ID, TEID). A user needing multiple QoS classes can set up different GTP tunnels towards the data network. The information about the QoS class / TEID is shared between modules using an eBPF `PERCPU` map and is provided by the *GTP Handler* in the uplink direction and by the *Classifier* in the downlink path. Despite our simplification, the implementation of the full 5G QoS model with multiple QoS flows per GTP tunnel does not require architectural changes, since the *Classifier* is already able to classify packets at flow level.

Following sections provide a more detailed description of each different module.

B. GTP Handler

In the upstream direction (UE-to-MGW) this module acts as a GTP tunnel terminator; it removes the GTP headers (i.e., GTP, UDP and outer IP) and retrieves the Tunnel Endpoint Identifier (TEID), which is then passed to the next module in the chain (i.e., *Traffic Policer*) through a shared eBPF `PERCPU` hash map. On the downstream direction (MGW-to-UE), it matches the IP destination address of the packet (i.e., the IP address of the UE) with an eBPF `HASH` map containing the UE-BS mapping. Then, it encapsulates the packet into a new GTP tunnel, retrieving the TEID from the shared eBPF map, and sends it to the base station.

C. QoS Management

The next module provides a way to enforce the required QoS thanks to its ability to *drop*, *pass* or *limit* the packets of a specific traffic class. For bandwidth management, we implemented and evaluated three different policing mechanisms in order to determine the best trade-off between complexity (hence, performance penalty) and performance. All have reduced memory overhead since they are bufferless, and they have small CPU overhead because there is no need to schedule or manage queues. More complex traffic shapers (e.g., pacing, hierarchical token bucket) are not entirely implementable in eBPF/XDP due to its event-based model, and require a cooperation with the Linux Traffic Control (TC) subsystem for buffer management and queuing.

Fixed Window Counter (FWC): Once a new packet is received, the MGW atomically decreases the Window Counter (WC) size in the map based on the packet size; when the value is zero the packet is discarded. A user space thread is in charge of resetting, every W seconds an eBPF `HASH` map containing the mapping TEID - WC, which is defined as the product of the desired rate R and the window size W. This is the simplest rate limiter, with a lightweight and fast data plane, albeit with some limitations. It is not possible to independently configure the average rate and the maximum burst size, since once one of these parameters is defined the other one is dictated by the size of the window. This size moreover cannot be too small due to the need of the user-space thread to periodically scan the counters associated to all QoS flows (potentially hundreds of thousands), and this may produce a coarse and bursty traffic.

Token Bucket (TB): To be forwarded, each packet needs to consume a number of tokens equal to its size. The bucket is refilled at a rate equal to the desired average rate, while its capacity represents the maximum burst. Unlike the *Fixed Window Counter* the refill of the bucket in user space is not a viable solution, since eBPF does not provide an adequate synchronization primitive between user and kernel threads. In fact, only map update operations are guaranteed to be atomic in user space, but this is not enough as the following racing condition can occur, affecting the precision of the TB and resulting in an output rate higher than the desired one:

1) The user space reads the current value of the bucket and computes the new number of tokens based on the refill rate and the maximum capacity.
2) At the same time, multiple packets are forwarded in the kernel, consuming tokens.
3) The user space writes the new value of the bucket in the map, hiding the tokens consumed in the former step.

To solve the above problem, we perform the bucket refill directly in the data plane: every bucket is associated with the timestamp of its last refill and tokens are optionally added on every packet reception. The `bpf_spin_lock()` and `bpf_spin_unlock()` eBPF helpers allow to update each bucket atomically.

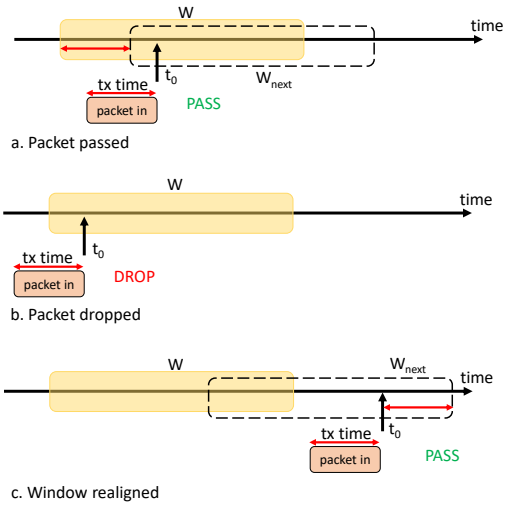

Figure 2: Sliding Window scenarios.

Sliding Window (SW) [6]: Given the rate limit of R and burst limit of B, a window of size $W = B/R$ is defined (i.e. the time needed to transmit an entire burst at the desired rate). Every time a new packet arrives, the time needed to transmit it at the desired rate is computed: for a packet of size S bits, $T = S/R$. In order to transmit the packet we must be able to shift forward the sliding window of a time T without exceeding the arrival time of the packet. The three possible scenarios shown in fig. 2 may occur:

(a) The arrival time of the packet falls in the window and its distance from the begin time of the window is bigger than the transmit time T: we pass the packet and move the window forward of T.
(b) The arrival time of the packet falls before (on the left) the window or its distance from the begin time of the window is smaller than the transmit time T: we drop the packet and do not touch the window.
(c) The arrival time of the packet falls after (on the right) the window: this means that we have not moved the window for too long (due to the absence of received packets). In this case we realign the end of the window to the arrival time and the shift it forward of a time interval T.

Also in this case (such as for the TB) we update the position of the window in the data plane and use spin locks to guarantee atomic operations.

D. Traffic Classifier

This module is used to map a packet in the downlink direction to its corresponding TEID, which is used to enforce the correct QoS and to perform GTP encapsulation. To support more complex classification rules we used the same algorithm defined in [7]. The Linear Bit Vector Search classification algorithm is compatible with the limited number of data structures available in eBPF and allows to speed up the classification process exploiting the parallelism of CPU registers, while maintaining a linear cost. The eBPF code is dynamically generated every time the configuration of the service changes, in order to include only parsing of needed headers and perform lookups only on the protocol headers actually used for the classification.

E. Router

The router component can work in both "shared" mode, where the host FIB table is used to decide the next hop of the packet through the Internet, or in "private" mode where a separate BPF `LPM_TRIE` map is used and configured by the MGW control plane. For the rest, no novel algorithms or implementation details are worth mentioning in this paper.

III. Evaluation

We compared our eBPF MGW with equivalent pipelines based on different data plane technologies (BESS [8], OvS-DPDK and OvS-kernel [9])[2] available in TIPSY [10], a benchmark suite to evaluate and compare the performance of programmable data plane technologies over a set of standard scenarios rooted in telecommunications practice. Where not differently specified, we performed all throughput tests according to RFC2544, tuning the input rate in order to obtain a packet loss lower than 1%, and using 64 bytes frames, since packet size turned out not to affect the results.

A. Rate limiting algorithms

We tested the algorithms proposed in section II-C to evaluate both their accuracy and the impact on performance.

Accuracy: For UDP traffic we generated packets at a high rate (40 Mpps) using MoonGen and fed them to the DUT, obtaining an almost perfect output rate in all cases, provided that the burst limit was big enough (for algorithms requiring it[3]). We used `iperf3` to evaluate the effect of the algorithms on the TCP protocol and used the *Token Bucket Filter* (`tbf`) queuing discipline of the kernel as a reference. Results in fig. 3 show that the *Token Bucket* is not able to produce the desired rate if it is configured with a burst limit smaller than the desired rate. This behaviour is due to the fact that the TCP protocol assumes (huge) intermediate buffers in mind, which have to be "emulated" by our bufferless solution by increasing the burst size. To prove this theory we emulated a bufferless behaviour with the (vanilla) Linux `tbf` by configuring a queue size of one packet and the results show that the qdisc is not able to produce the desired rate as well. We obtained a similar behavior by testing different rate limits and using the *Fixed Window Counter* and the *Sliding Window* algorithms.

Overhead: One of the key features of an eBPF MGW is the ability to leverage all the CPU cores provided by the machine, since the traffic reaching the server is usually distributed on different cores by Receive Side Scaling (RSS) based on the 5-tuple of the packet. The *Traffic Policer* is the most critical

[2]Tester and Device Under Testing (DUT) are connected with a dual-port Intel XL710 40Gbps NIC. DUT has an Intel Xeon Gold 5120 14-cores CPU @2.20GHz (hyper-threading disabled) and Ubuntu 18.04.1 LTS. Moongen packet generator. Kernel 5.9 for eBPF, kernel 5.0 with DPDK 19.11 for other technologies.

[3]With a millisecond clock resolution a burst bigger than 1/1000th of the desired rate is required.

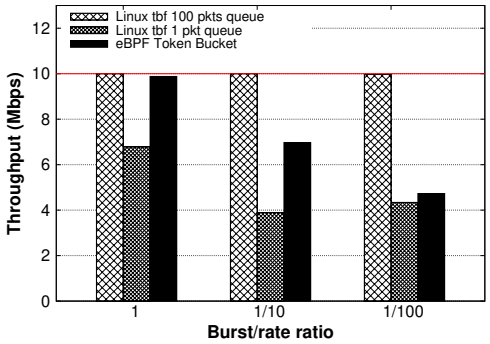

Figure 3: Output rate with 10 Mbps rate limit.

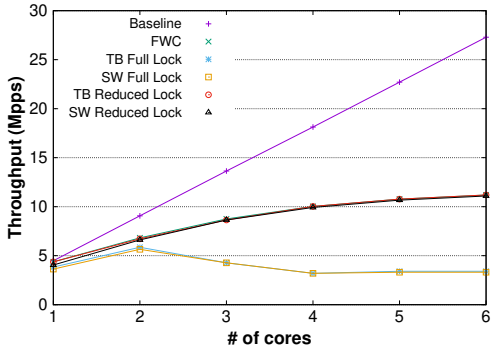

Figure 4: Rate limiters on multiple cores with a single traffic class.

module for what concerns multi-core scalability since multiple flows belonging to the same QoS class could be processed concurrently on different cores and try to access the same window/bucket. In this scenario the naive use of spinlocks, covering all the rate limiting portion of code, could become a bottleneck. Fig 4 exacerbates the problem trying to police all the traffic processed by different cores in the same QoS flow. On the other hand, the Fixed Window Counter, which relies only on an atomic increment operation to update its counter (`sync_fetch_and_add()`), is able to scale, even if not in a linear way. In order to reduce the usage of spinlocks in the other two algorithms we made the following observations:

- In the *Token Bucket* the spinlock is only needed when refilling the bucket, since we need to atomically add the tokens and update the timestamp of last refill. Tokens can be consumed by packets with the atomic increment operation.
- In the *Sliding Window* the spinlock is only needed in scenario (*c*), when reattaching the window to the current time, since we need to prevent multiple cores from overwriting their reattach operation. Vice versa, Moving forward the window in scenario (*a*) can be done with an atomic increment operation.
- The maximum rate of execution of the two operations listed above is bound to the time resolution we use.

We restricted the use of spinlocks to the sections specified above and chosen a millisecond time resolution, that should limit the execution of locked sections of code while

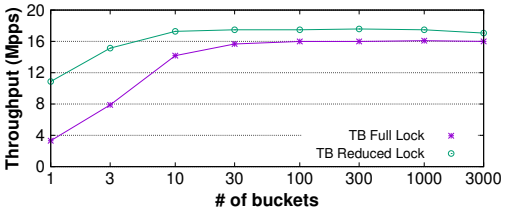

Figure 5: Rate limiting on six cores with different numbers of buckets.

keeping a good precision. This also allows us to replace the `bpf_ktime_get_ns()` helper, whose overhead proved to be non-negligible, with a custom clock stored in a `PERCPU_ARRAY` map and updated every millisecond by a thread in user space. We obtained values similar to the *Fixed Window Counter* as shown by the *Reduced Lock* in fig. 4.

We evaluated the impact of the cross-cores interference discussed above by steering the traffic on six different cores and increasing the number of buckets used to handle that traffic. Results in fig. 5 show that the effect of cross-cores interference becomes negligible when traffic is managed by more than 100 buckets, however our improved solution still retains a performance boost of about 10% since it avoids the overhead needed to acquire and release spinlocks.

B. Scalability with multiple users

We scaled the number of configured users (each one with a single tunnel) up to 3000, setting one base station every 100 users and one additional route on the PDN every 10 users. We configured Moongen to generate an average of 10 UDP flows per user. Fig. 6 shows that, in the downlink direction, the eBPF pipeline outperforms both the in-kernel alternative and also (user space) BESS with a high number of configured users, due to the poor scalability of the latter (w.r.t [2] section *V-B*), while OvS-DPDK still retains a high performance lead. This changes in the uplink direction shown in fig. 7: here the eBPF pipeline does not need to classify packets (an expensive operation whose cost grows linearly with the number of users) and its throughput is more consistent. While in the downlink direction the OvS-DPDK pipeline relies on the Linux kernel to perform routing, in uplink it uses its internal, less optimized, longest-prefix-matching algorithm (w.r.t [2] section *V-A*), resulting in a higher performance drop with an increasing number of users.

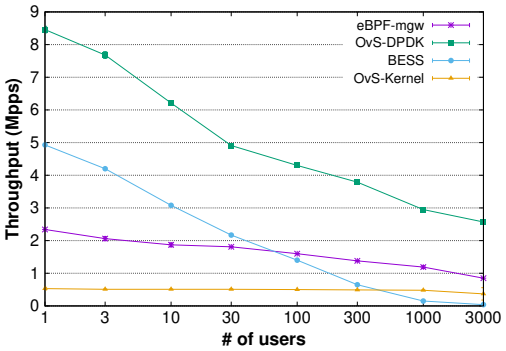

Figure 6: Multiple users scalability (downlink).

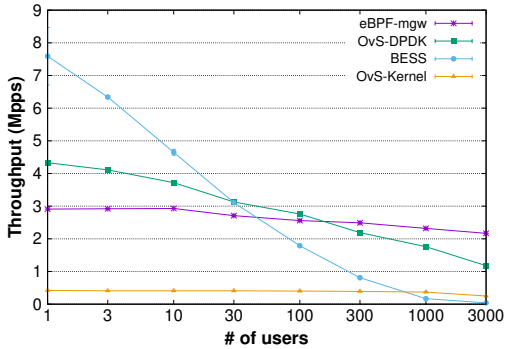

Figure 7: Multiple users scalability (uplink).

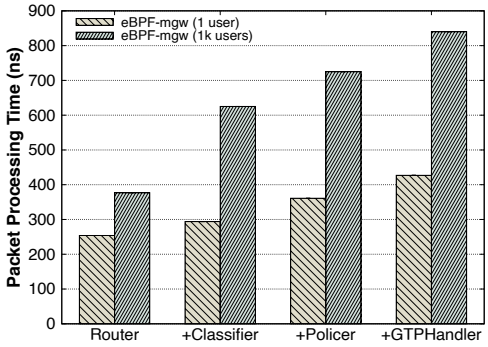

Figure 9: Packet processing time breakdown.

C. Multicore scalability

We configured a base of 100 users, 10 routes and 1 base station per core, increasing the number of cores used to process the traffic, generating again an average of 10 flows per user. Fig. 8 shows that the scalability of the eBPF implementation is in line with the one of its in-kernel and user space counterparts. We omit the results of BESS since its throughput seems to decrease even adding more cores to computation.

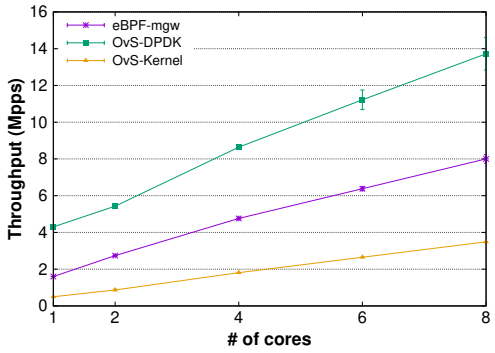

Figure 8: Multicore scalability (downlink).

D. Modules overhead

We analyzed the impact of the different modules on the performance of the eBPF gateway with both low (1) and high (1000) number of configured users. Fig. 9 shows the average time needed to process each packet, starting with the *Router* module alone and then adding the others. Results show that the most resource-hungry service is the *Classifier*, whose algorithm scales linearly with the number of rules we use in this scenario. However, we feel that this can be reduced with a more careful implementation.

IV. Conclusions

In this paper we presented a proof-of-concept 5G Mobile Gateway based on the eBPF and XDP technologies and made a point for its use in scenarios with limited resources such as Edge Computing, where servers need to be shared between network related tasks and generic workloads. We prototyped a simplified architecture and showed the limitations imposed by eBPF in its implementation as well possible solutions. While our solution proposes a completely in-kernel implementation of the network function, the introduction of the `AF_XDP` socket type opens the possibility to perform some of the more complex tasks in user space while avoiding the drawbacks of traditional kernel-bypass technologies [11]. We leave a careful evaluation of this technology to our future work. Our evaluation and comparison with other technologies shows that eBPF is an interesting alternative, especially in those cases where some performance can be sacrificed in exchange for a higher integration with the kernel and a more flexible resource usage.

References

[1] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th international conference on emerging networking experiments and technologies*, 2018, pp. 54–66.

[2] T. Lévai, G. Pongrácz, P. Megyesi, P. Vörös, S. Laki, F. Németh, and G. Rétvári, "The price for programmability in the software data plane: The vendor perspective," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2621–2630, 2018.

[3] S. K. Singh, C. E. Rothenberg, G. Patra, and G. Pongracz, "Offloading virtual evolved packet gateway user plane functions to a programmable asic," in *Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms*, 2019, pp. 9–14.

[4] D. Lee, J. Park, C. Hiremath, J. Mangan, and M. Lynch, "Towards achieving high performance in 5g mobile packet core's user plane function," Intel Corporation, SK Telecom, Tech. Rep., 2018.

[5] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A framework for ebpf-based network functions in an era of microservices," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 133–151, 2021.

[6] Q. Monnet. (2017) Stateful packet processing: two-color token-bucket poc in bpf. [Online]. Available: https://github.com/qmonnet/tbpoc-bpf

[7] S. Miano, M. Bertrone, F. Risso, M. Bernal, Y. Lu, and J. Pi, "Securing linux with a faster and scalable iptables," 2019.

[8] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," 2015.

[9] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[10] T. Authors. (2018) Tipsy: Telco pipeline benchmarking system. [Online]. Available: https://github.com/hsnlab/tipsy

[11] N. Van Tu, J.-H. Yoo, and J. W.-K. Hong, "Accelerating virtual network functions with fast-slow path architecture using express data path," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1474–1486, 2020.