Fastlane: A framework for building fast path network applications

Debojeet Das, Lakshya, Mythili Vutukuru
Department of Computer Science and Engineering, Indian Institute of Technology Bombay
Email: {debojeetdas, lakshya, mythili}@cse.iitb.ac.in

***Abstract*—To leverage the full potential of high-speed network interface cards (NICs), it is essential for network applications to use high-performance packet I/O frameworks. However, porting applications to use these specialized frameworks is a tedious task. While prior work has proposed frameworks to ease the development of network functions over fast packet I/O mechanisms, there has not been much work done to ease the porting of applications that are built to run on the kernel network stack. This paper presents Fastlane, a framework to easily port the "fast path" of traditional network applications to use fast packet I/O mechanisms while retaining the "slow path" to run on the traditional network stack. Fastlane provides APIs by which application developers can configure, use, and switch between different fast packet I/O mechanisms like DPDK and CNDP in the application's fast path. Additionally, Fastlane provides APIs for the fast path to communicate with the slow path running on the kernel network stack. Fastlane also allows applications to be deployed in an unprivileged Kubernetes pod, easing application development for cloud deployments. We evaluate the effectiveness of Fastlane by porting a production-grade 5G user plane function to use our framework. Our evaluation shows that Fastlane enables developers to build high-performance network applications easily and imposes minimal performance overheads.**

***Index Terms*—Network Function Virtualization, Middleboxes, Cellular Networks, Zero-Copy Networking**

I. Introduction

High-speed network interface cards (NICs) offer bandwidths of tens or hundreds of gigabits per second. However, traditional network stacks are inefficient for such speeds [1], especially in cloud environments where virtualized networks require packet traversal through both host and container stacks [2]. These stacks were originally designed for Layer 7 applications and are not optimized for Layer 2 and Layer 3 network functions, like L4 firewalls and L3 load balancers.

Many high-speed applications use kernel-bypass or in-kernel-offload techniques such as Data Plane Development Kit (DPDK) [3] or eXpress Data Path (XDP) [4] to address these inefficiencies. In cloud environments, these techniques are paired with hardware virtualization methods like Single Root I/O Virtualization (SR-IOV) [5] to achieve performance comparable to bare-metal deployments [6]. However, porting applications built on traditional network I/O to these frameworks can be complex and require significant development effort [7]. Moreover, in multi-tenant cloud deployments, configuring fast I/O paths may require privileges that are not available.

Prior work [8]–[12] has focused on easing the development of applications over fast packet I/O frameworks but has primarily addressed L2/L3 or L4/L7 network functions. However, applications like the 5G User Plane Function (UPF) require both L2/L3 processing (for routing packets using GTP header manipulation) and L4/L7 processing (for communicating with control plane nodes over HTTP/TCP and UDP). In most UPFs used today, there exists a *fast path* which handles the high-speed L2/L3 user traffic using optimized frameworks like DPDK, while the *slow path* uses the kernel's TCP/IP stack for lower-rate L4/L7 communication. Porting such complex applications to fast I/O mechanisms requires significant manual code changes. Cloud environments add further challenges, as setting up the fast path may require privileged operations, and choosing between packet I/O mechanisms may not be obvious, requiring substantial rewrites. To the best of our knowledge, no prior work has eased this porting effort for complex applications like the UPF that span L2/L3 and L4/L7 functionalities.

This paper introduces Fastlane, a framework that enables developers to port traditional network applications to fast I/O mechanisms while retaining complex slow-path processing on the kernel TCP/IP stack. The implementation of our framework supports porting to two fast I/O mechanisms: DPDK and the CNDP [13] that is based on AF_XDP [14]. Fastlane abstracts out the setup and usage details of these fast I/O mechanisms with simple APIs and configuration files, allowing application developers to port to and switch between the fast I/O mechanisms with only changes to the configuration file. Furthermore, Fastlane provides APIs for easy communication between fast and slow paths and manages privileged packet I/O operations in cloud environments through a Kubernetes device plugin, allowing unprivileged containers to run ported applications seamlessly.

To demonstrate Fastlane's utility, we ported a basic network application that swaps L2 headers (macswap) and UPF from a production-grade 5G packet core [15]. With minimal code changes (less than $\sim$ 10%), the traditional UPF seamlessly transitioned to operate as the slow path. In contrast, the new Fastlane API-based code, which reused the UPF packet processing logic from the traditional socket API, facilitated the implementation of a fast path. Performance measurements show that Fastlane introduces minimal performance overhead, making it a practical solution for high-speed packet processing applications. Fastlane is completely open-source and can be found at https://github.com/rickydebojeet/fastlane.git.

979-8-3503-8053-8/24/1.00 ©2024 IEEE

II. Background & Related Work

Fast packet I/O mechanisms. The Linux kernel network stack is not optimized for high-speed packet processing due to bottlenecks like packet copying, context switching, and bulky data structure allocations [8]. Kernel bypass techniques like DPDK resolve this by allowing direct, zero-copy access to NIC packets in userspace, offering high performance but with downsides such as high CPU usage and reduced compatibility with kernel tools.

Understanding the requirement of high-speed packet processing, the Linux kernel introduced extended Berkeley Packet Filter (eBPF) [16], enabling custom userspace programs to run in a sandboxed, event-driven environment. The XDP hook [4], located in the driver, allows efficient packet handling before entering the kernel stack, enabling actions like dropping, transmitting, or redirecting to AF_XDP sockets. AF_XDP offers an alternative to kernel bypass by enabling zero-copy packet redirection to userspace using XDP, balancing performance and CPU utilization. It supports both event-driven and busy-polling modes, with the latter offering better performance at the cost of higher CPU usage [17].

Cloud Native Data Plane (CNDP) [13], built on AF_XDP, accelerates network applications without custom drivers or memory pinning, ensuring compatibility with cloud orchestration platforms like Kubernetes [18]. DPDK also supports AF_XDP as an alternative to its poll-mode drivers but requires memory pinning and huge pages.

So, which of these different fast I/O packet mechanisms must an application use? This question has no clear, correct answer, and each mechanism has its own pros and cons. Busy-polling modes generally provide higher performance, but interrupt-based AF_XDP modes are more CPU efficient, especially in cloud environments. We compare the performance of DPDK and AF_XDP-based CNDP in different polling modes using a simple "macswap" application running on a single CPU. Figure 1 compares throughput when different polling modes (userspace denoted as US , polling denoted as POLL, or busy polling denoted as BP) were used in DPDK and CNDP. Figure 2 shows CPU usage as a function of load in a different experiment for some subset of these fast I/O mechanisms. Our findings show that DPDK's userspace polling achieved the highest throughput, with AF_XDP-based CNDP offering lower performance but better CPU efficiency under low loads. Thus, the optimal mechanism varies with the deployment context, and applications should be adaptable to different mechanisms with minimal code changes.

Cloud deployments. When deploying network applications in the cloud on containers or VMs, packet traversal through two network stacks can be bypassed using hardware virtualization techniques like SR-IOV. SR-IOV allows containers to share physical NIC resources by passing them directly as Virtual Functions (VFs). This enables multiple containers with a DPDK application to run inside them over SR-IOV VFs. However, configuring VFs requires privileges, as does running eBPF-based AF_XDP applications. Kubernetes device plugins

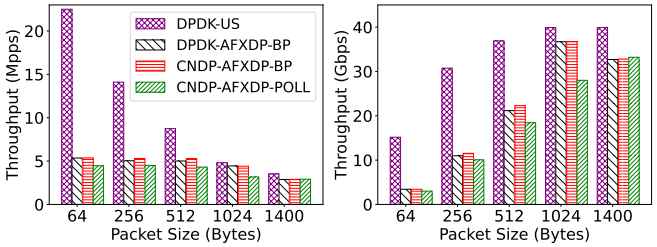


Figure 1: Performance comparison of DPDK vs. CNDP

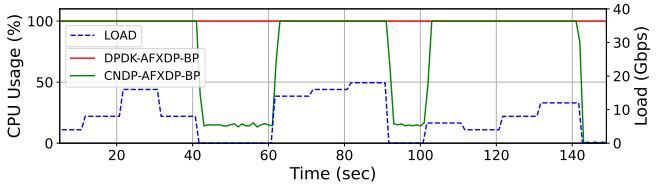


Figure 2: CPU usage of DPDK vs. CNDP

for SR-IOV [19] and AF_XDP [20] enable unprivileged containers to offload these privileged operations to the plugin, allowing fast network I/O applications to run in unprivileged environments. Hence, to support Fastlane, this setup needs to be extended to support our fast-path, slow-path design.

5G User Plane Function. The 5G mobile packet core connects the user equipment (UE) and the radio components to external networks and consists of a control plane and a data plane. The control plane is responsible for managing UE connections, and the data plane forwards UE traffic. When a UE wants to send data, it sets up one or more "sessions" in the packet core. The packet forwarding rules corresponding to these sessions are sent to the UPF via the Packet Forwarding Control Protocol (PFCP) by the control plane. These rules contain Packet Detection Rules (PDRs) to find matching packets, Forward Action Rules (FARs), and other rules that tell the UPF how to handle the matching packets. When a UE sends or receives data to and from an external network, the IP datagrams are encapsulated in GPRS Tunnelling Protocol (GTP) headers and transmitted over UDP to the UPF, which then uses the rules received from the control plane to suitably route the packet.

Related work. Many network applications leverage DPDK for its kernel-bypass benefits [21], [22], but porting traditional network stack applications to DPDK requires substantial effort. To address this, frameworks like OpenNetVM [9] provide abstractions that simplify building L2/L3 network functions over DPDK. Other frameworks [10], [12] support high-performance applications using XDP by filtering packets for L2/L3 applications and passing others to the kernel stack. Nethuns [23] offers common abstractions for different network I/O mechanisms but lacks support for DPDK and cloud-native deployments. LemonNFV [24] consolidates various fast network I/O techniques but, unlike Fastlane, does not enable easy porting of complex functions like the UPF to multiple packet I/O mechanisms across both bare-metal and cloud environments.

For L4-L7 network functions, userspace TCP stacks like mTCP can be used alongside the above L2/L3 frameworks for fast packet I/O, although these stacks cannot match the

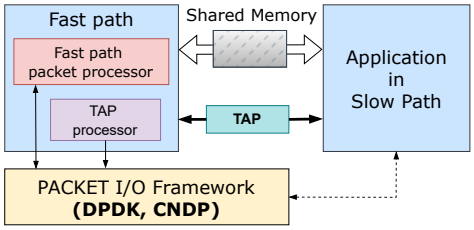


Figure 3: Fastlane Design

functionality of the Linux network stack. Thus, it is desirable to allow L4/L7 functions that do not require high performance to run natively on the Linux stack, a feature supported by Fastlane.

Middlenet [11] is closest to Fastlane, building L2/L3 and L4/L7 functions on the same machine using SR-IOV VFs. However, Fastlane addresses a slightly different problem, where the same application has an L2/L3 fast path and an L4/L7 slow path and where application developers wish to port existing applications easily to fast I/O mechanisms and switch between different packet I/O mechanisms with no code changes. Junction [7] offers compatibility for unmodified apps but lacks support for multiple packet I/O mechanisms.

III. System Design & Implementation

Fastlane has been designed to meet the following goals:

- The developer should be able to port applications written for traditional kernel stack to the Fastlane framework with minimal changes.
- The API provided by the library should be agnostic to the underlying packet I/O framework so that the developer can write code to use or switch between fast packet I/O mechanisms without worrying about their internals.
- The library should provide mechanisms to communicate and share state between the fast and slow paths.
- The library should allow deployments in cloud platforms like Kubernetes without requiring privileged operations within the container.

Figure 3 shows the high-level architecture of a Fastlane application, consisting of independent fast and slow path components, with the former running on a fast packet I/O mechanism like DPDK or CNDP and the latter on the Linux kernel. Fastlane allows the setup of a TUN/TAP device and optional shared memory to exchange messages and share application states between the fast and slow paths. For example, in the case of UPF, the packet forwarding rules are shared between the fast and slow paths using the shared memory.

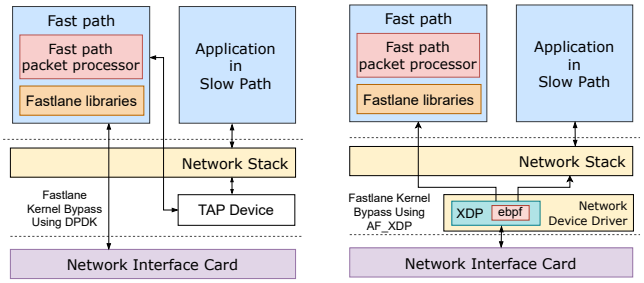
The application developer writes a JSON *configuration file* to specify several parameters of the fast packet I/O mechanism, including the framework to use for the fast path, the details of the logical ports (which can be PCIe address of physical devices or SR-IOV VFs for DPDK, and interface for CNDP) and its queues which are used for the fast path, the port filters that identify fast path traffic from slow path traffic (e.g., UDP port number on which GTP traffic is received at the UPF) and many other such parameters. The Fastlane framework reads the configuration file and seamlessly sets up the plumbing for each packet I/O mechanism. Packets destined for the fast path are suitably filtered in the hardware via NIC filtering rules or in software via the eBPF program at the XDP hook and delivered to the fast path packet processor. The remaining packets are injected into the kernel stack and reach the slow path.

Fastlane currently supports DPDK (high performance but high CPU utilization) and CNDP (AF_XDP based, more compatible with Linux tools, but lower performance) packet I/O mechanisms, but support for more packet I/O methods can be added easily. Figure 4 shows two different example setups of the packet I/O path based on different DPDK and CNDP configurations. In Figure 4(a) using DPDK, packet filtering is set in a way that all packets are delivered to the fast path, and packets destined to the slow path are injected into the kernel using a TUN/TAP device by the fast path code itself. In Figure 4(b) using CNDP, the packet filtering and redirection are handled in an eBPF program running at the XDP hook of the device driver. The fast path packets are sent to the AF_XDP program, and the slow path packets are passed on to the traditional network stack. Note that these are just two example configurations of the packet I/O path, and several other possible configurations can be easily realized via Fastlane using its rich set of parameters exposed in the JSON configuration file with no additional development effort.

In addition to specifying configurations, Fastlane developers must also write (or port) fast path and slow path components using the Fastlane APIs. Fastlane provides developers with a comprehensive suite of APIs for writing fast-path packet processing code, including managing framework setup, shared memory allocation, inter-path communication, and packet buffer operations. These APIs are agnostic to the packet I/O mechanism, and Fastlane invokes suitable code in the DPDK, CNDP, or other packet I/O libraries automatically. Slow path code can be written using traditional kernel APIs, but Fastlane APIs must be used to communicate with the fast path.

Fastlane supports building multithreaded fast path applications with ease. Our library supports two models for the packet processing callback invocation via the threads: run-to-completion and master-worker. In the run-to-completion model, Fastlane spawns one processor thread for each queue in the logical port, invoking the callback function upon packet reception and handling transmission if needed. In the master-worker model, the master processor threads manage packet reception and transmission. When a packet is received, the master threads distribute the packets to the worker threads for application processing.

To address the challenge of running a DPDK or CNDP-based fast path packet processor within an unprivileged container, Fastlane sets up a device plugin to configure the privileged operations for the packet processor. For AF_XDP, the device plugin performs operations like XDP program loading and XSK map creation, which then passes the map file descriptor to the fast path packet processor via a Unix Domain Socket (UDS). The device driver also makes similar modifications to set up the SR-IOV VF, which can be used by the DPDK application running inside the container. To support



(a) DPDK Internal Design (b) AF_XDP Internal Design

Figure 4: Fastlane internal implementation

our fast path/slow path model, we have used the plugin to configure the TAP device on behalf of the Fastlane application.

To showcase the benefits of Fastlane, we have ported 5G UPF obtained from a production-grade 5G packet core implementation [15], built on the kernel network stack, to Fastlane. We rewrote the GTP packet processing fast path of the UPF using Fastlane APIs so that the data plane operation of forwarding user traffic happens over a fast packet I/O mechanism. The remaining part of the UPF, which registers with the mobile packet core, participates in node discovery, and exchanges packet forwarding rules with the 5G packet core control plane, was left to run on the kernel network stack as the slow path. Note that the slow path processes complex HTTP/JSON messages over REST APIs, and porting this code to a fast packet I/O mechanism is difficult. Neither was it necessary, given the low traffic rate the slow path is expected to handle. We made minimal changes to the slow path to share packet forwarding rules and other application states using shared memory. Overall, we needed to change only about 472 lines of code (out of 4954 lines in the main UPF packet processing logic) to port the UPF to act as a slow path for Fastlane. Additionally, 311 lines of code were modified in the GTP packet processing logic from the slow path to create the new fast path using Fastlane APIs.

IV. Evaluation

In our evaluation, we first measure the performance overhead of using Fastlane. We also evaluate the multicore scalability that Fastlane provides. Finally, we evaluate the performance of the 5G UPF ported to Fastlane.

The System Under Test (SUT) running Fastlane applications was equipped with Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz, 128 GB RAM, and Intel XL710 NICs with 40 Gbps line rate. The SUT was running on Ubuntu 22.04 with Linux kernel 5.15.0-25-generic; the SUT was optimized with hyperthreading disabled and CPUs explicitly isolated for the application's execution. The underlying frameworks were configured to use 2MB hugepages to improve Translation Lookaside Buffer (TLB) misses, ensuring optimal performance. To generate the load, we used a DPDK-based Pktgen traffic generator [22], which generates traffic using six RX-TX queue pairs with a burst size of 32 packets. For the evaluation of Fastlane DPDK deployment, we have only used custom user space polling drivers, and for the Fastlane CNDP deployment, we have used the busy polling mechanism.

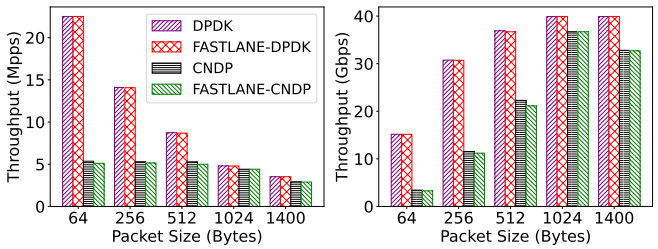


Figure 5: Throughput of macswap operation

Framework	Latency (μs)		
	64B	256B	512B
DPDK	13.34	15.15	18.76
Fastlane-DPDK	13.56	15.24	18.86
CNDP	33.54	33.68	35.21
Fastlane-CNDP	36.95	37.12	37.21

Table I: Latency of macswap operation

A. Overhead and multicore scalability

We first perform microbenchmarks to analyze the Fastlane overhead and ensure that Fastlane applications do not perform any worse than when executed natively using DPDK or CNDP. We generated line rate traffic from the load generator to a Fastlane application that performs "macswap" operations while running on a single core, with the packet then sent back to the load generator. Similarly, we used native CNDP and DPDK-based "macswap" applications and tested the maximum achievable throughput of the application with less than 1% packet drops. We maintained identical configurations and setups across Fastlane and native deployments to facilitate a fair comparison. The latency was also noted to ensure that Fastlane does not add significant processing time to the native processing time of the framework. Note that we only present fast path performance results and do not focus on the performance of the slow path because the slow path running on the Linux kernel is not expected to have a very high throughput in any case. We also omit results of the performance of Fastlane applications running inside containers, as the findings and conclusions were similar to the baremetal experiments described below.

Figure 5 shows the comparison of the throughput of the macswap network function when running with and without Fastlane. Table I shows the corresponding end-to-end latency. The results show that DPDK and our Fastlane-DPDK implementation achieved a line rate for packet sizes of 1024B and 1400B. Similarly, CNDP and Fastlane-CNDP exhibited nearly identical performance levels. However, all fast packet I/O versions performed significantly better than the same application running on the Linux kernel stack, which achieved a throughput of only 55 Mbps for 64B packets and 498 Mbps for 1400B packets. Our results show that applications built over Fastlane perform similarly to those built directly over the fast packet I/O mechanisms like DPDK or AF_XDP. Also, the performance overhead of Fastlane in terms of throughput or latency is minimal.

Next, we measure the throughput of the same application using 128B packets and with increasing CPUs. Figure 6 shows

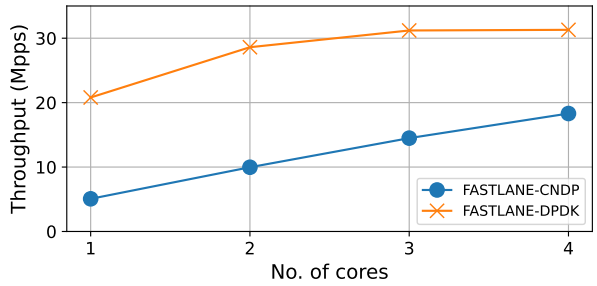


Figure 6: Scaling of macswap operation

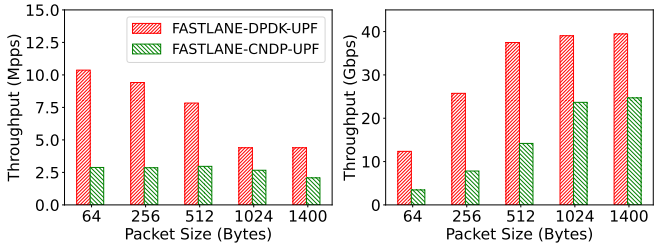


Figure 7: Throughput of 5G UPF using Fastlane

the throughput of the macswap application with increasing cores. We see that the throughput of the macswap application increased almost linearly until it reached the line rate of the NIC, showing that Fastlane allows easy development of multicore scalable fast path applications.

B. 5G UPF with Fastlane

To evaluate the performance of the Fastlane UPF, we performed load testing within the proprietary production-grade 5G core implementation environment, simulating a scenario with 10K concurrent 5G users transmitting data to the 5G core, with 20K PDRs and 20K FARs stored in the UPF. Figure 7 depicts the performance of the UPF after it was ported to Fastlane, with four cores in the fast path and the slow path running the original single-threaded UPF with minimal changes. For 1024B and 1400B packets, Fastlane's DPDK-based UPF achieved a line rate performance, whereas Fastlane's CNDP-based UPF achieved $\sim$ 2.9 Mpps. The results show that Fastlane is useful for easily porting the fast path of complex network applications like UPF to fast packet I/O mechanisms while allowing the slow path to remain on the kernel.

V. Conclusion & Future Work

This paper introduces Fastlane, a framework designed to port the fast path of network applications to use a high-performance packet I/O mechanism while allowing the slow path to continue running on the traditional network stack with minimal modifications. Application developers can use Fastlane APIs, which are agnostic to the underlying packet I/O mechanism, requiring only minimal changes to their applications. They can switch between packet I/O mechanisms by modifying a configuration file without altering the application code. Fastlane abstracts the complexities of packet path setup and internal details of packet I/O mechanisms, managing them seamlessly in both bare-metal and cloud environments platforms like Kubernetes. Our evaluation demonstrates that complex applications like the 5G UPF can be easily ported to Fastlane with negligible overhead. We plan to integrate Fastlane with the auto-scaling framework in Kubernetes and explore the possibility of switching from DPDK to CNDP framework at runtime based on resource availability.

References

[1] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *ACM SIGCOMM*, 2021.

[2] R. Nakamura, Y. Sekiya, and H. Tazaki, "Grafting sockets for fast container networking," in *ANCS*, 2018.

[3] dpdk.org, "Intel Data Plane Development Kit." http://dpdk.org/, 2018.

[4] 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 *ACM CoNEXT*, 2018.

[5] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," *Journal of Parallel and Distributed Computing*, Nov. 2012.

[6] N. Shyamkumar, P. Raczynski, D. Cremins, M. Kubiak, and A. S. Rajan, "In-kernel fast path performance for containers running telecom workloads," *Netdevconf 0x16*, 2022.

[7] J. Fried, G. I. Chaudhry, E. Saurez, E. Choukse, I. Goiri, S. Elnikety, R. Fonseca, and A. Belay, "Making kernel bypass practical for the cloud with junction," in *NSDI*, 2024.

[8] M. Abranches and E. Keller, "A Userspace Transport Stack Doesn't Have to Mean Losing Linux Processing," in *NFV-SDN*, 2020.

[9] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "OpenNetVM: A Platform for High Performance Network Service Chains," in *HotMIddlebox*, 2016.

[10] 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*, 2021.

[11] S. Qi, Z. Zeng, L. Monis, and K. K. Ramakrishnan, "MiddleNet: A Unified, High-Performance NFV and Middlebox Framework With eBPF and DPDK," *IEEE Transactions on Network and Service Management*, 2023.

[12] N. Van Tu, J.-H. Yoo, and J. Won-Ki Hong, "Accelerating Virtual Network Functions With Fast-Slow Path Architecture Using eXpress Data Path," *IEEE Transactions on Network and Service Management*, 2020.

[13] M. Karlsson, G. Loughnane, E. Mathew, P. Osikoya, J. Shaw, M. Tahhan, E. Verplanke, and K. Wiles, "Cloud Native Data Plane (CNDP) - Overview," *Technology Guide*, 2024.

[14] M. Karlsson and B. Topel, "The Path to DPDK Speeds for AF XDP," *LPC Net*, 2018.

[15] "Proprietary 5g core implementation." Indigenuous 5G testbed project, IIT Bombay, 2021.

[16] M. Fleming, "A thorough introduction to ebpf." https://lwn.net/Articles/740157/, 2024.

[17] F. Parola, R. Procopio, R. Querio, and F. Risso, "Comparing User Space and In-Kernel Packet Processing for Edge Data Centers," *ACM SIGCOMM CCR*, 2023.

[18] K8s, "Kubernetes." https://kubernetes.io/, 2024.

[19] "SRIOV network device plugin for Kubernetes." https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin, 2024.

[20] I. Corporation, "intel/afxdp-plugins-for-kubernetes." https://github.com/intel/afxdp-plugins-for-kubernetes, 2024.

[21] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "Mtcp: A highly scalable user-level tcp stack for multicore systems," in *NSDI*, 2014.

[22] pktgen, "Pktgen—traffic generator powered by dpdk.." https://github.com/pktgen/Pktgen-DPDK, 2024.

[23] N. Bonelli, F. D. Vigna, A. Fais, G. Lettieri, and G. Procissi, "Programming socket-independent network functions with nethuns," *ACM SIGCOMM CCR*, 2022.

[24] H. Li, Y. Dang, G. Sun, G. Liu, D. Shan, and P. Zhang, "LemonNFV: Consolidating heterogeneous network functions at line speed," in *NSDI*, 2023.