
Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering ($36^{th}$cycle)

# Orchestrating Edge Computing Services with Efficient Data Planes

By

**Federico Parola**

******

**Supervisor(s):**
Prof. Fulvio Risso

**Doctoral Examination Committee:**
Prof. Dejan Kostic, Referee, KTH Royal Institute of Technology (SE)
Prof. Gabor Retvari, Referee, Budapest University of Technology and Economics (HU)
Prof. Mario Baldi, AMD Research (US)
Prof. Guido Marchetto, Politecnico di Torino (IT)
Prof. K. K. Ramakrishnan, University of California Riverside (US)

Politecnico di Torino
2024

# Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Federico Parola
2024

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

# Acknowledgements

This thesis marks the end of a journey that shaped the way I am, both professionally and personally. A rich journey full of ups and downs, moments of enthusiasm and hard work, and moments of impasse, many successes and satisfactions and some failures. Overall, this PhD was a successful and enriching experience, and many people contributed to this achievement.

First of all, I'd like to thank my supervisor, Fulvio Risso, for introducing me to the world of scientific research and guiding me during these years. He taught me to think out of the box and pursue my ideas without giving up even in front of setbacks, and was always there to give me advice when I most needed it.

A thank also goes to TIM S.p.A. for sponsoring my PhD, and to Roberto Procopio and Roberto Querio, my TIM supervisors, for the fruitful exchange we had during the PhD.

I'd like to thank all the people from lab 9, with whom I shared many days of work. It was always a pleasure to have lunch together and chat and laugh to relieve the pressure of work.

Special thanks go to the people from the UCR, Professor K.K. Ramakrishnan, people from the computer networks laboratory, people from the international scholars group, and my housemates, they all helped make my time in California an awesome experience, both professionally and personally, and always made me feel at home.

Thanks to all my friends with whom I shared many happy experiences during these years which were always a pleasant and usually much-needed distraction from work.

Last but not least, my deepest thanks to my family, in particular my parents and my brother, who supported me during this journey despite all the ups and downs and were always there to listen to my complaints and give me good advice.

# Abstract

In the rapidly evolving landscape of telecommunications, the advent of the 5G technology has emerged as a revolutionary force, promising to redefine the way we connect, communicate, and experience the digital world. One of the architectural cornerstones enabling 5G innovation is represented by Multi-access Edge Computing (MEC), which involves deploying computing resources at the edge of the network, closer to the end-users and devices, guaranteeing lower latencies and reducing the amount of data that needs to traverse the network backbone. This paradigm shift is coupled with the almost complete migration to Network Function Virtualization (NFV) for networking tasks which, by moving traditional networking functions from dedicated hardware appliances to software-based solutions, provides greater flexibility, scalability, and efficiency in the deployment and operation of network services.

The transfer of an increasing share of computing workloads to small edge deployments, coupled with the execution of network functions on the same general-purpose servers, pose non-trivial challenges in how edge data centers are managed. This makes resource consumption, an already strategic topic in the cloud context, an even more crucial requisite to make edge computing an effective solution. Hence, in this dissertation, we analyze the problem of resource optimization in edge data centers, with the goal of maximizing the number of applications and network functions that can be deployed in the constrained network edge and benefit from its improved performance. We do so by focusing on three main optimization areas. We start by studying how to enable the **consolidation of cloud and network workloads on the same physical server** as the first building block for a more efficient and flexible edge. Since kernel-bypass frameworks widely used for packet processing (e.g., DPDK) pose non-negligible drawbacks in terms of integration and resource-sharing with traditional kernel-based cloud applications, we explore the capabilities recently introduced by the eBPF and XDP technologies to perform fast packet processing in

the kernel, as well as to flexibly redirect packets to user space without completely bypassing the former. We show how, thanks to these technologies, a combination of in-kernel and user space packet processing can be leveraged to maximize the performance of both cloud native and network workloads. Then, we move to **enabling secure and efficient multi-tenancy** on edge servers, aiming to facilitate safe access to this precious resource. To this end, we design a virtualization environment based on unikernels, to combine the leanness of containers with the strong isolation of virtual machines, and integrate it with a high-performance zero-copy data plane to handle inter-service communication. Finally, we focus on the increasing demands that interconnecting application modules poses on data center networking, both in terms of performance and flexibility. In this respect, we design ways of **improving the management of services through eBPF**, for both traditional cloud workloads and chains of network functions. Focusing on Kubernetes, the most widespread cloud orchestrator, we design an alternative to classic, monolithic eBPF-based networking providers, proposing an approach based on modular, reusable building blocks coming from the traditional networking world, such as routers, bridges, and load balancers, which despite its higher level or abstraction does not compromise performance. We also enable applying the automatic scaling capabilities of Kubernetes to network function chains, leveraging a mechanism of flexible cross-connections to efficiently interconnect replicas of network functions.

Overall, the work presented in this dissertation contributes to enhancing the capabilities of edge data centers to support the ever-increasing load they have to handle, with a high degree of efficiency, making the promises of 5G closer to becoming reality.

# Contents

# List of Figures

# List of Tables

# Part I

# Introduction

# Chapter 1

# Introduction

The advent of the 5G technology has emerged as a revolutionary force in the landscape of telecommunications, promising to redefine the way we connect, communicate, and experience the digital world. The fifth generation of mobile network represents a significant leap forward from its predecessors, offering unprecedented speed, reliability, and capacity [5]. 5G's exponential increase in speed does not only enable quicker downloads of large data and high-definition content but also opens the door to a multitude of innovative applications, such as augmented reality (AR) and virtual reality (VR). Furthermore, 5G boasts remarkably low latency, reducing the delay between sending and receiving data. This low latency is crucial for applications that demand real-time responsiveness, such as autonomous vehicles and remote surgery. 5G is also designed to support a vast number of connected devices simultaneously. This improved capacity is particularly critical as we transition into the era of the Internet of Things (IoT), where everyday objects are interconnected to exchange data seamlessly. The enhanced connectivity of 5G can accommodate the growing number of IoT devices, enabling smart cities, smart homes, and a more interconnected world [6].

Fulfilling this set of promises and the stringent requirements that follow calls for a complete architectural redesign of 5G networks with respect to their predecessors. One of the cornerstones of this architecture redesign is represented by Multi-access Edge Computing (MEC) [7]. MEC involves deploying computing resources at the edge of the network, closer to the end-users and devices, rather than relying solely on centralized cloud data centers. MEC significantly reduces latency by bringing

computing resources closer to the point of data generation and consumption. This is crucial for applications that demand ultra-low latency, such as augmented and virtual reality, autonomous vehicles, and real-time industrial automation. By processing data at the edge, MEC minimizes the time it takes for data to travel between the device and the centralized cloud, resulting in faster response times. Moreover, by offloading data processing tasks from the central cloud to local edge servers, MEC helps reducing the amount of data that needs to traverse the network backbone, optimizing bandwidth usage and improving overall network efficiency, a fundamental feature given the huge capacity that 5G foresees to provide. MEC architecture also enables scalability and flexibility in deploying computing resources. By distributing computing capabilities across multiple edge locations, the network can scale more efficiently to handle increasing data traffic and the growing number of connected devices. Concerning privacy, edge computing allows sensitive data to be processed locally, reducing the need to send raw, sensitive information to centralized cloud servers. This enhances security, as critical data stays closer to its source and is subject to localized security measures [8]. It also minimizes the exposure of data during transit over the network.

Another key enabler of 5G innovation is represented by the almost complete migration to Network Function Virtualization (NFV). NFV is a concept that shifts traditional networking functions from dedicated hardware appliances to software-based solutions, providing greater flexibility, scalability, and efficiency in the deployment and operation of network services. At its core, NFV involves decoupling network functions from proprietary hardware and implementing them in software. This abstraction allows network services to run on standard servers, storage, and switches, eliminating the need for specialized, purpose-built hardware. With the ability to deploy and scale network functions through software, service providers can adapt quickly to changing demands without the need for significant hardware changes. Moreover, by virtualizing network functions, NFV enables efficient resource utilization, with multiple VNFs that can run on a shared physical infrastructure, optimizing the use of computing resources and reducing the overall hardware footprint. Virtualized network functions can be dynamically scaled to meet changing demand, ensuring optimal service quality and performance. This leads to cost savings and improved energy efficiency. Last but not least, NFV accelerates the deployment of new network services: since functions are implemented in software, the time and effort required to roll out updates are significantly reduced.

The transfer of an increasing share of computing workloads to small edge deployments, combined with the execution on general-purpose servers of networking functions previously relegated to dedicated hardware, poses non-trivial challenges in how edge data centers are designed and operated. Compared to the traditional cloud computing environment, which can rely on big, centralized data centers with plenty of available resources, edge computing is based on a multitude of small, distributed data centers, which should provide the same functionalities of cloud deployments while relying on a limited number of servers. This makes resource consumption, already a strategic topic in the cloud context, an even more crucial aspect to make edge computing an effective solution. The goal of leveraging resources efficiently is challenged by different aspects that characterize the edge scenario.

In the first place, while the cloud infrastructure is commonly known to adapt elastically to load and leverage resources efficiently, the integration of cloud-native workloads with data plane network functions still remains an open problem given the different nature of these two workloads. Cloud-native applications are usually structured as a mesh of loosely-coupled services following the microservices-based approach. This multitude of services, running in containers or Virtual Machines (VMs), leverage the Linux networking stack to communicate with each other to provide the final service, with orchestration tools such as Kubernetes interacting with the Linux kernel to handle resource allocation and provide connectivity. However, while providing a widespread and consolidated abstraction layer to build feature-rich applications, the Linux network stack is known to yield sub-optimal performance in the case of packet processing applications [9], that may need to handle traffic rates up to 150 Mpps per interface in the case of 100 Gbps Ethernet links [10]. In this context where every nanosecond matters, the generality of the Linux kernel, which allows it to support a multitude of applications, comes at a price that usually cannot be afforded. As a consequence, the data plane VNF scenario mainly relies on kernel bypass frameworks, to provide the packet processing application with direct access to the network card, hence cutting away all the costs related to Linux. Frameworks such as DPDK [11], netmap [9], and PF_RING ZC [12] provide direct access to the network hardware from userspace and allow to build highly specialized network stacks focused on performance. Multiple individual network functions as well as higher-level VNF frameworks rely on these solutions or other forms of kernel bypass as the base for their performance [13, 14, 15, 16]. Nevertheless, kernel bypass solutions present some limitations when it comes to integration with cloud-native

workloads. First, they are usually resource-hungry, requiring the static allocation of a set of cores dedicated to busy polling on NIC queues to receive packets avoiding the cost of interrupts. This however does not mix nicely with the resource-sharing needs of cloud deployments, and even less in the resource-constrained edge scenario. Second, they either require completely detaching the network interface from the control of the Linux kernel or the use of custom drivers. Custom drivers might not be available in every environment, and exclusive control of the NIC prevents services leveraging the traditional Linux TCP/IP stack from accessing the network. Solutions to the second problem exist but have their limitations as well. Re-injection of traffic in the kernel from user space usually comes with poor performance (as also shown in this dissertation, Chapter 3), while user-space implementations of the TCP/IP stack [17, 18] require changes to application code and do not provide all the tools (e.g., `iptables`, `tc`) and security guarantees that the well-tested Linux kernel carries with itself.

Recently, the introduction of a novel technology in the Linux kernel, eBPF [19], has opened a third alternative to packet processing. The Extended Berkeley Packet Filter (eBPF) is a general-purpose in-kernel virtual machine that enables the execution of custom programs within the kernel space. This allows developers to efficiently extend and customize kernel functionality without the need for kernel code modifications. eBPF programs are designed to be memory-safe and run in a restricted environment, preventing them from causing harm to the system. This safety is enforced by a verifier that checks the program's integrity before it is loaded into the kernel, ensuring that it adheres to a set of predefined rules and constraints. eBPF programs can be attached to different hook points inside the Linux kernel, in order to react to different events such as system call executions, file access, packet reception and transmission, and so on, providing a useful tool in the fields of observability, security, and networking. In the latter field, the introduction of a high-performance hook point, the eXpress Data Path (XDP) [20], has helped to close the gap with kernel bypass solutions. XDP allows the processing of packets at the earliest possible stage in the Linux kernel, at the driver level, before expensive operations such as the allocation of the `sk_buff` structure are performed. This allows achieving high performance in scenarios where packets are dropped (e.g., firewalling and DDoS mitigation [21, 22]) or redirected out of the machine (e.g., load balancing [23]), while still being able to yield traffic to the Linux TCP/IP stack.

The different networking needs of data plane network functions and cloud-native workloads is not the only obstacle to the efficient and flexible resource sharing required by edge computing. Multi-tenancy is a key component to guarantee that edge resources can be leveraged by different operators that, for cost-related reasons, might not be able to reach the level of extensiveness required by edge computing with their proprietary resources. When multi-tenancy comes into play, isolation between the workloads of different customers becomes a key concern. However, given that revenues for edge providers are directly connected to the number of services they can pack on their servers, this isolation cannot come at the cost of performance and resource consumption. Traditional virtualization solutions, Virtual Machines and containers, stand at the edges of the performance-isolation spectrum, with the first providing a strong level of isolation at the cost of a higher resource footprint, and the second being more lightweight and flexible albeit with weaker security guarantees [24].

Given the new demands introduced by edge computing and its tight resource budget, this dissertation faces the problem of resource optimization in the context of edge computing with a focus on the networking component. We study the applicability of both kernel-based and kernel-bypass approaches when facing the problem of efficient resource sharing between cloud-native workloads and packet processing network functions, as well as providing efficient chaining mechanisms between them.

## 1.1 Summary of contributions

This dissertation is structured into three main topics that translate into the three central parts of the work. The first part deals with the coexistence of cloud and network workloads on the same physical server, to address the resource constraints of edge data centers. To do so we explore the potential of in-kernel packet processing, enabled by eBPF and XDP, in supporting telco-level network functions, and then combine it with user-space alternatives to achieve a balance of high performance for packet processing applications and good integration with kernel-based cloud-native applications. We then move to the topic of multi-tenancy to allow efficient allocation of edge resources to customers. We do so by focusing on the increasingly widespread serverless computing paradigm, and by designing a framework that

allows efficient sharing of server resources between tenants, by supporting high-speed communication between application modules without compromises on isolation and security. To achieve this goal we focus mainly on kernel-bypass technologies, by envisioning a scenario in which the whole edge infrastructure can be migrated to our framework. On the other hand, in the last part, we focus on supporting the microservices design for vanilla kernel-based applications, by providing efficient and straightforward service chaining facilities leveraging the powers of eBPF, targeting both the interconnection of cloud-native applications and network functions.

We introduce in the following the above contributions, with a greather level of detail.

### 1.1.1 Enabling efficient coexistence of cloud and network workloads on the same physical server

Since the edge is in charge of providing the network-related functionalities by mobile users to access the Internet (e.g., routing, QoS, security), as well as running offloaded cloud workloads, being able to efficiently co-schedule cloud applications and packet processing network functions is a key component to achieve optimal resource utilization. When starting this study, however, both research-oriented and production-grade telco network functions operating on the user plane were based on DPDK or other kernel-bypass technologies, a choice driven by the high performance enabled by these frameworks, albeit at the cost of higher resource consumption. Still, the rigid resource partitioning required by these technologies, such as dedicated CPU cores and network interfaces, can end up in wasted resources. In this respect, eBPF/XDP looks like a more appealing solution, thanks to its capability to process packets in the kernel, achieving a low-overhead integration with non-data plane applications relying on the kernel TCP/IP stack, albeit with lower packet processing performance than DPDK. Hence, we start the dissertation by exploring the applicability of eBPF and its high-performance packet processing hook, XDP, to telco-grade network functions. To do so, we focus on a 5G User Plane Function (UPF), the key network function of the 5G core network in charge of, as the name suggests, handling the user plane. We propose the first proof-of-concept open-source implementation of a 5G UPF based on eBPF/XDP, highlighting the possible challenges (e.g., to create traffic policers, as buffering is not available in eBPF) and the resulting architecture. To verify its

applicability to real scenarios, we characterize it in terms of performance and scalability and compare it with alternative technologies, showing that it outperforms other in-kernel solutions (e.g., Open vSwitch) and is comparable with DPDK-based alternatives.

Given the promising results of this first work, we take a step forward and dig deeper into the capabilities offered by the XDP Linux subsystem. We analyze AF_XDP, a new family of Linux sockets that allows to efficiently steer packets in user space. Unlike "traditional" kernel-bypass technologies such as DPDK, AF_XDP allows keeping the NIC under the control of the kernel, with an XDP program in charge of selecting which packet should be sent to user space and which not. We leverage the XDP/AF_XDP combo to provide a thorough comparison of user space vs in-kernel packet processing, focusing on how these two approaches can be leveraged and optionally combined to address the network traffic mix that traverses an edge data center, composed of *pass-through* traffic, processed by a chain of NFs and redirected to a remote destination on the internet, *local* traffic, directed to applications running locally (i.e., offloaded on the same edge node running the network function), and *dropped* traffic, which has to be discarded e.g., for security reasons. Our results provide useful insights on how to select and combine these technologies to improve overall throughput and optimize resource usage.

### 1.1.2 Enabling secure and efficient execution of serverless workloads on multi-tenant servers

To maximize the number of players that can take advantage of edge computing capabilities, edge providers strive to pack as many workloads as possible on a limited set of physical nodes. Serverless computing maps nicely to this attempt, with customers providing only their business logic (usually in the form of functions, following the *Function-as-a-Service* approach) and the cloud (or edge) operators in charge of handling resource allocation, connectivity, security, and other management aspects. This provides more opportunities for resource optimization, with the operator orchestrating applications of all customers in a centralized way, instead of each customer handling its pool of (usually over-provisioned) resources. On the other hand, running unknown workloads of different, potentially competing customers on the same shared infrastructure requires leveraging sandboxing mechanisms stronger than

plain containers, which fail to provide proper security and isolation in a multi-tenant environment. However, mechanisms such as VM-based execution or other sandboxes (e.g., Google's gVisor) introduce non-trivial overheads when compared to the already taxing containerized runtime. This incentivizes the exploration of a serverless design that is both secure and lightweight. To address this need we describe SURE, a unikernel-based serverless framework with a high-performance and secure data plane. Unlike in the previous part, where we address support for unmodified applications leveraging the Linux TCP/IP stack, our design of a custom virtualization runtime pushes us toward an architecture based mainly on kernel bypass. SURE's data plane supports both intra-node and inter-node zero-copy communication. The first leverages the reliable nature of inter-process communication within a single host to provide a high-performance data plane based on shared memory buffers exchange, to move messages without needing additional protocols such as TCP/IP. The second leverages a user-space zero-copy TCP/IP stack (Z-stack), that seamlessly interacts with the shared memory processing. Additionally, SURE provides library-based sidecars to establish a lightweight service mesh. Unlike traditional service mesh solutions that rely on an individual userspace sidecar for each application instance, this approach provides the same monitoring and enforcement capabilities at a fraction of the cost. SURE employs a group-based security domain design that strictly isolates shared memory processing between untrusted functions, incorporating access control enabled by sidecars and Z-stack. To tackle the security concerns of shared memory, we leverage Intel's Memory Protection Keys (MPK) to ensure safe access to the data plane and library-based sidecar from untrusted application code, while preserving the efficient single-address-space nature of unikernels. These combined efforts create a more secure and robust data plane while improving throughput up to 17 times over a traditional approach, based on containers and gRPC interconnections.

### 1.1.3 Improving the management of services through eBPF

In the last part, we go back to supporting standard kernel-based applications and notice how, within the data center, the interconnection between application modules themselves provides opportunities for optimization. Indeed, one of the strengths of the widespread microservice application design is its ability to flexibly and granularly scale each module composing the full application by creating or removing replicas, to adapt to application load. This however makes networking within a data center

increasingly complex, with the need to efficiently interconnect application modules (or *pods* in Kubernetes terminology), whose number and location can change at a high rate. In Kubernetes, the most widespread microservices orchestrator, the task of handling networking is carried out by an external component, the network provider. In this context, eBPF has already found a widespread application, with multiple eBPF-based network providers existing. However, we notice how the most prominent examples of eBPF network services follow a monolithic approach, in which all required code is created within the same program block. This makes the code hard to maintain and extend, and difficult to reuse in other use cases, especially for network operators used to traditional networking components such as bridges and routers. To address this concern we leverage the Polycube framework to demonstrate that a disaggregated approach is feasible also with eBPF, with minimal overhead, introducing a larger degree of code reusability. We design and implement a complete Kubernetes network provider based on elementary basic blocks coming from the traditional networking world (e.g., bridge, router, firewall, NAT, load balancer, and more) and evaluate its performance showing no penalties with respect to existing solutions.

Aside from the management of traditional applications, the microservice approach is gaining increasing attention from network operators, who want to leverage its benefits for the management of their packet processing network functions. Indeed, service function chains, ordered sets of network functions that provide network services to the handled traffic, need to adapt to traffic which is usually highly variable over time. Although Kubernetes has already brought benefits to general-purpose applications, it is not natively suitable for network workloads since it lacks some functionalities required by network services. We close this gap by designing a simple, cloud-native architecture that integrates SFCs in Kubernetes, to seamlessly leverage cloud-native features such as horizontal autoscaling. The solution is based on flexible cross-connections, namely logical links that connect adjacent network functions, which can promptly adapt the distribution of the network traffic to the existing network functions in case of scale in/out events affecting the number of NF instances. We validate the solution with an open-source proof-of-concept implementation based on dedicated Kubernetes operators and an eBPF load balancer, demonstrating the feasibility and the efficiency of the proposed approach.

### 1.1.4 Previously published work

This thesis includes previously published and co-authored works. In particular, Chapter 2 and Chapter 3 are adapted from the works presented in [25] and [26], while Chapter 5 and Chapter 6 are an adaptation of [27] and [28]. Chapter 4 is adapted from a work not yet published, carried out in collaboration with the research group of prof. K. K. Ramakrishnan at University of California, Riverside.

# Part II

# Enabling efficient coexistence of cloud and network workloads on the same physical server

# Chapter 2

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

## 2.1 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 [20], 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.

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 [29]. Other works like [30] and [31] 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 chapter 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.

## 2.2 Design

Figure 2.1 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 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.

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

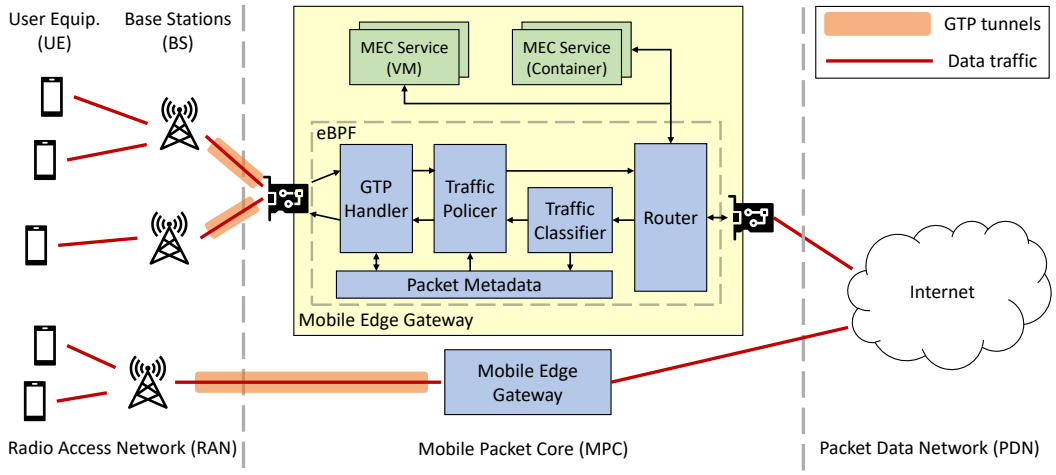


Fig. 2.1 Mobile Gateway prototype architecture.

### 2.2.1 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 [32] 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 (Section 2.3), 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.

### 2.2.2 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.

### 2.2.3 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.

*Sliding Window (SW) [33]*: 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 Figure 2.2 may occur:

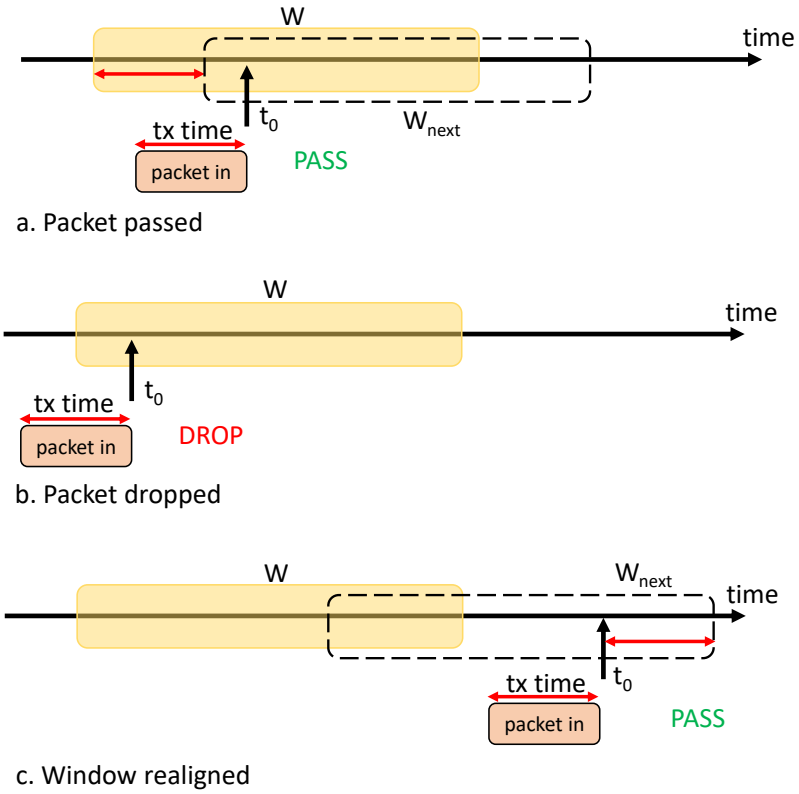


Fig. 2.2 Sliding Window scenarios.

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.

### 2.2.4 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 [21]. 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.

### 2.2.5 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 chapter.

## 2.3 Evaluation

We compared our eBPF MGW with equivalent pipelines based on different data plane technologies (BESS [34], OvS-DPDK and OvS-kernel [35])[2] available in TIPSY [36], 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.

[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.

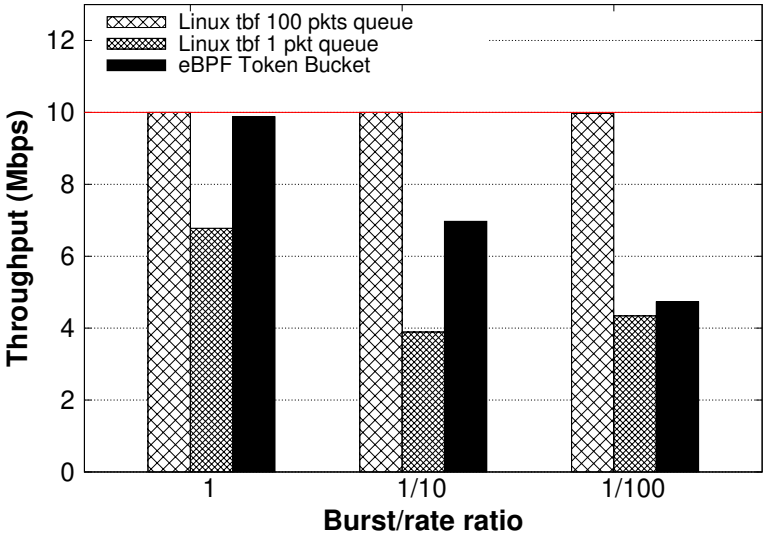


Fig. 2.3 Output rate with 10 Mbps rate limit.

### 2.3.1 Rate limiting algorithms

We tested the algorithms proposed in Section 2.2.3 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 Figure 2.3 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.

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

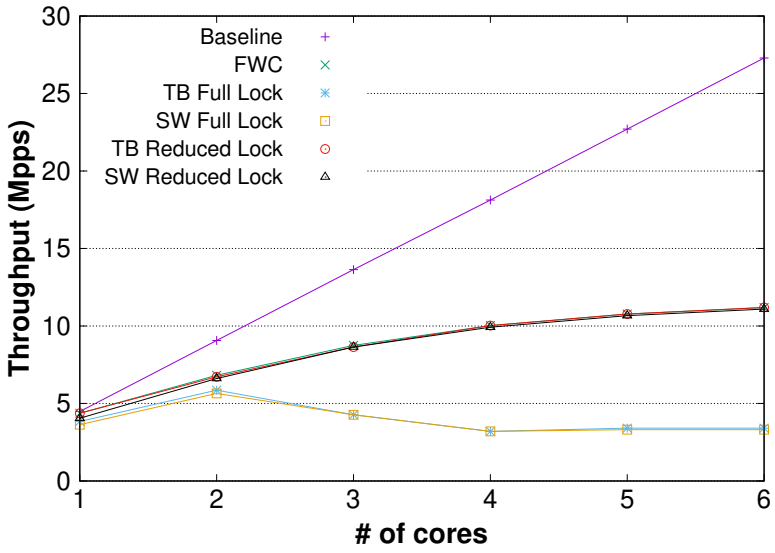


Fig. 2.4 Rate limiters on multiple cores with a single traffic class.

*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 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. Figure 2.4 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

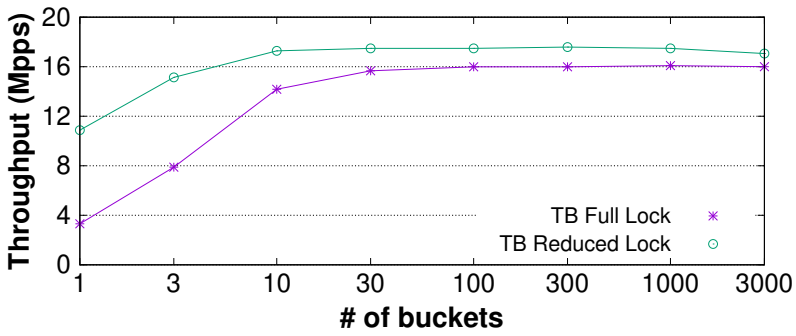


Fig. 2.5 Rate limiting on six cores with different numbers of buckets.

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 chose a millisecond time resolution, that should limit the execution of locked sections of code while 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 Figure 2.4.

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 Figure 2.5 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.

### 2.3.2 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

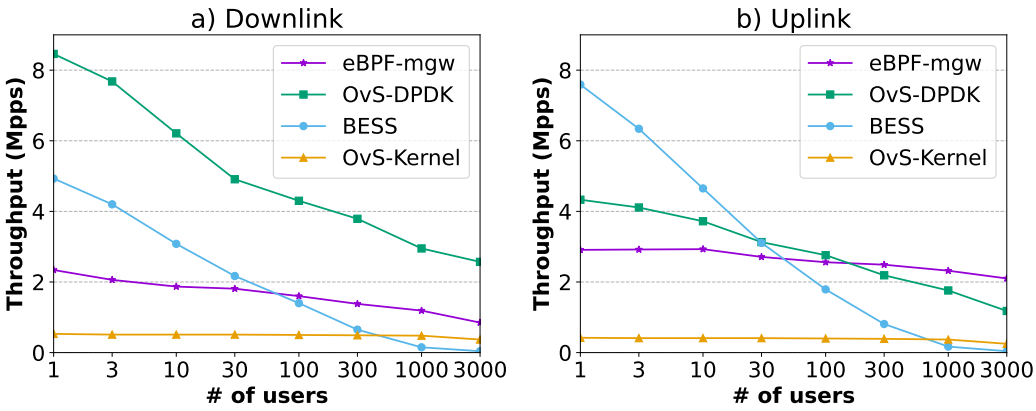


Fig. 2.6 Multiple users scalability, downlink (a) and uplink (b).

per user. Figure 2.6-a 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 [29] section *V-B*), while OvS-DPDK still retains a high-performance lead. This changes in the uplink direction shown in Figure 2.6-b: 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 [29] section *V-A*), resulting in a higher performance drop with an increasing number of users.

### 2.3.3 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. Figure 2.7 shows that the scalability of the eBPF implementation is in line with that 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.

### 2.3.4 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. Figure 2.8

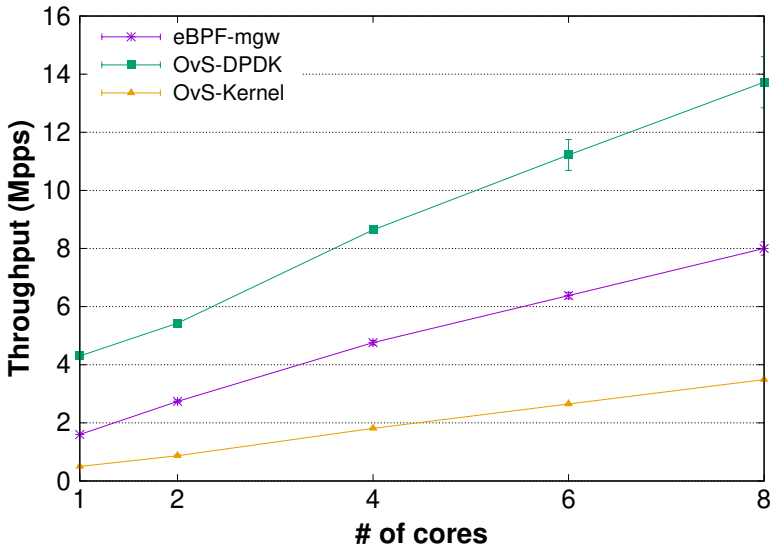


Fig. 2.7 Multicore scalability (downlink).

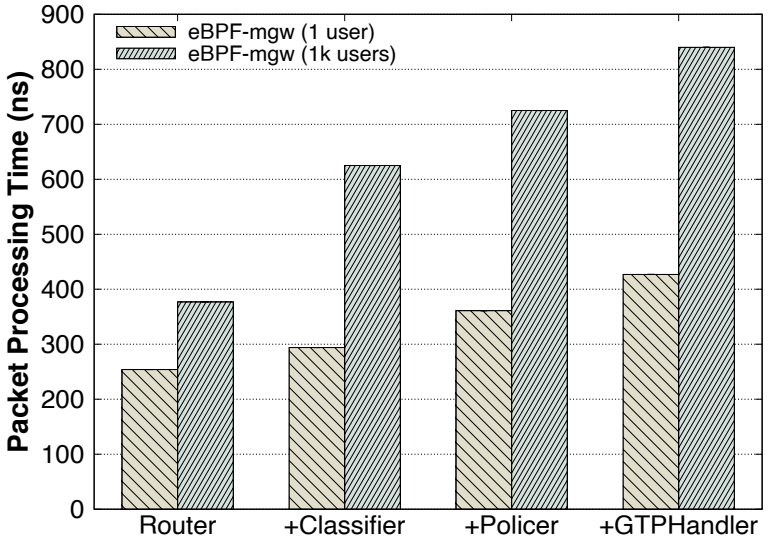


Fig. 2.8 Packet processing time breakdown.

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.

## 2.4 Conclusions

In this chapter, 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 as 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 [37]. 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.

# Chapter 3

# Comparing User Space and In-Kernel Packet Processing for Edge Data Centers

## 3.1 Introduction

Edge computing is a paradigm that moves computational capabilities close to the end user, in order to provide services with lower latency and increase the amount of available bandwidth. Unlike cloud computing, where user data is processed in big, centralized data centers with almost unlimited resources, processing at the edge requires telco operators to support a large number of small, distributed data centers, each one featuring a few servers. Traffic of the user reaching these data centers has to be processed by a fixed chain of Network Functions (NFs) that provide basic connectivity to the global network (e.g. 5G User Plane Functions, a.k.a. UPFs, NATs). This traffic might be further processed by additional network services (e.g., firewalls) and can be either directed to the Internet or to applications running in the same data center (e.g., 5G control plane services, object recognition software, content caches, etc.), located at the edge for different reasons such as latency requirements, data aggregation, or resiliency concerns.

Traditionally, the above two types of workload (data plane NFs and traditional applications) are handled by partitioning available servers in two subsets. Data-plane workloads are usually executed on servers that leverage kernel-bypass packet

processing frameworks such as Intel DPDK. Instead, traditional applications are orchestrated by platforms such as Kubernetes and leverage the widespread Linux TCP/IP networking stack. In fact, kernel-bypass technologies allow to process packets in user space, completely avoiding the overheads introduced by the kernel network stack. They are well known for their flexibility and for providing very high throughput, but can hardly be executed in servers running also traditional applications due to the necessity to rely on rigid resource allocation schemes (e.g., CPU pinning with dedicated CPU cores, huge memory pages), and the difficulties to support applications that leverage the standard TCP/IP stack (which, in fact, needs to be re-implemented in user space [17]). On the other side, existing kernel-level network processing primitives (e.g., Netfilter, Traffic Control, etc.) are highly integrated with applications relying on the network stack, but introduce an unnecessary overhead (hence, low throughput) to pass-through traffic, that only needs NF processing.

While the approach based on rigid servers partitioning may be appropriate in cloud data centers, it may provide a sub-optimal resource usage when a few servers are available, which could severely impact the edge scenario. On one side, packets moving between NF and application servers generate additional traffic in the data center (east-west traffic) and require additional I/O operations. On the other side, the rigid partitioning of servers based on the type of application may lead to wasting some of the available resources. In this scenario, a shared approach that consolidates both workloads on the same machine(s) would enable a more efficient usage of resources, allowing to co-locate NFs and applications working on the same traffic and to allocate spare resources to any kind of workload.

In recent years, the introduction of the eXpress Data Path (XDP) [20] and AF_XDP has provided the missing ingredients to efficiently handle traffic either in kernel or in user space, on the same platform. XDP allows to process packets in the NIC driver [38, 39], retaining the possibility to yield a packet to the Linux network stack, while AF_XDP sockets can be used to bypass limitations of eBPF programs and provide flexible processing in user space. However, it is still unclear how to use the above technologies at the same time in a single server, to handle the complex processing scenario envisioned at the edge of the network.

This chapter presents the performance analysis of in-kernel and user space packet processing based on XDP/AF_XDP, including *pass-through* traffic, processed by a chain of NFs and redirected to a remote destination, *local* traffic, directed to

applications running locally, and *dropped* traffic, which has to be discarded e.g., for security reasons. This has the aim of determining which technology is best suited for each case, and to provide insights on how to optimally handle the mixed scenario typical of edge data centers.

While most recent network cards, such as Intel 800 series, provide customized hardware packet processing at the NIC level, we did not include this technology in our evaluation. Indeed, this chapter focuses on a fully software approach, which allows to be completely independent from the underlying hardware and supports the case (rather common at the time of writing) of the many data centers that feature network cards with limited packet processing capabilities (such as our Intel 700 series).

This chapter is structured as follows. Section 3.2 provides a background on the two main technologies considered in this chapter (namely, XDP and AF_XDP), as well as on HW/SW packet steering mechanisms that proved to be a key to further improve performance. Section 3.3 details the methodology and scenarios encompassed in our tests, while Sections 3.4 to 3.6 present the results of our experiments for the above tests scenarios, namely dropped, pass-through and local traffic. Section 3.7 combines the takeaways from previous experiments to provide guidelines to handle heterogeneous combinations of traffic and experimentally verifies their effectiveness. Section 3.8 reviews the literature related to the topic, while Section 3.9 draws the main conclusions.

## 3.2 Background

### 3.2.1 eBPF/XDP

eBPF is a virtual machine that allows the extensions of the functionalities of the kernel with custom code that can be injected at run time and executed at various hook points (e.g. trace points, system calls, every kernel function, etc.). eBPF programs leverage a special bytecode that is generated by the Clang/LLVM toolchain starting from a source code written in a (restricted) C language, which can be compiled just-in-time into native machine code for extra performance. Upon injection, eBPF programs are analyzed by a verifier, whose aim is to guarantee that the code cannot harm the kernel, for example checking that only allowed memory accesses are

performed and that the program will eventually terminate. As a consequence, eBPF programs have some limitations, such as a maximum number of instructions and the lack of support for unbounded loops. Moreover, they cannot access memory in a custom way, but need to rely on maps, a set of key-value stores with different access semantics (array, hash, queue, etc.), that can be shared between several eBPF programs and with the user space, and can be used to preserve the state among multiple executions of a program. Despite these limitations, eBPF has proven to be suitable to the creation of reasonably complex NFs, especially if limited to headers processing [40].

The eXpress Data Path provides a hook to execute high speed eBPF packet processing programs before actually entering the main Linux kernel. XDP allows, in its native mode, to execute these programs in the NIC driver, at the earliest possible point after a packet is received from the hardware, before the kernel allocates its per-packet `sk_buff` data structure or performs any parsing. Based upon the result of the processing, the program can either ask the kernel to drop the packet, let it continue for further processing in the network stack, send it back on the receiving interface or redirecting it to another net device or to user space thanks to AF_XDP sockets.

### 3.2.2 AF_XDP sockets

AF_XDP sockets are a new socket family that allows user space code to exchange packets with the NIC with very limited overhead, very similar to existing kernel-bypass technologies [41]. An application leveraging AF_XDP has to create an array of user-space memory called *UMEM*. The UMEM is a chunk of contiguous memory divided into equal-sized buffers, each one holding a single Ethernet frame. These buffers are used to move packets between the NIC driver and the user space application, exchanging their pointers through four circular rings allocated by the kernel. Figure 3.1 shows the lifecycle of a UMEM buffer in an application receiving packets from the network and dropping/redirecting them. At startup, buffers are waiting in the `fill` ring (1). When a packet arrives, the NIC stores it in a buffer available in the `fill` ring and moves its pointer to the `rx` ring (2), where it is consumed (i.e., processed) by the user space application (3). If the packet is dropped, its buffer is re-added to the `fill` ring (3a); otherwise it is queued to the `tx` ring for redirection (3b). The driver transmits packets from the `tx` ring and moves their

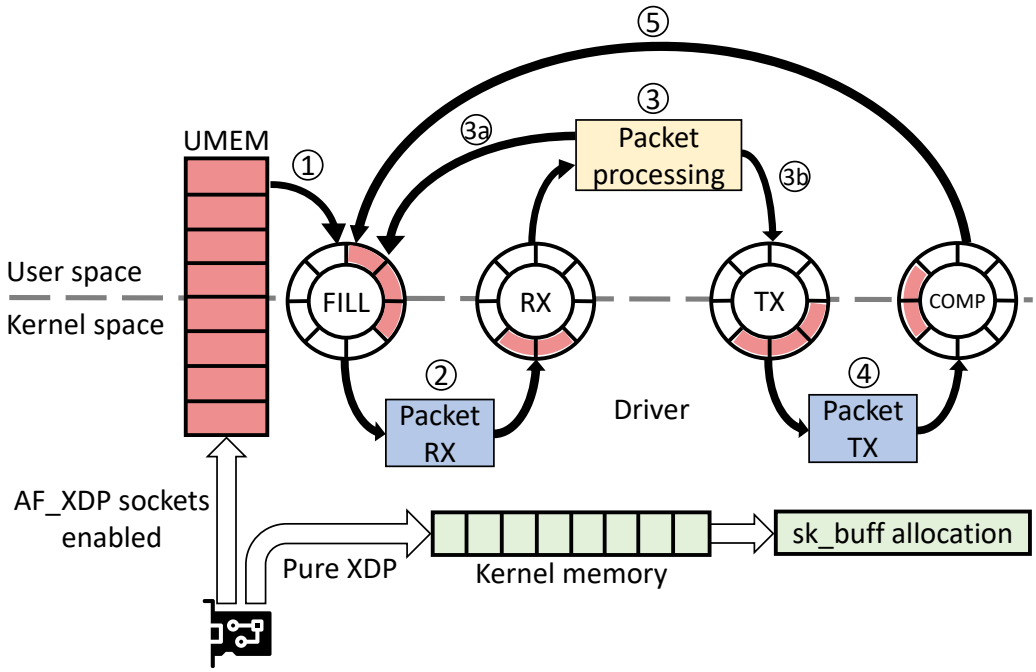


Fig. 3.1 The life cycle of a UMEM buffer in an application receiving traffic from the network.

buffers to the `comp` (i.e., *completion*) ring (4), leaving to the user space application the responsibility to move the above buffers back to the `fill` ring (5), ready to keep new packets.

Every AF_XDP socket is bound to a single `{netdev, queue}` couple (multiple sockets for the same couple are allowed), and is associated with one `rx` and one `tx` ring. A single UMEM can be shared between different sockets, however, a new pair of `fill` and `comp` rings is needed for every `{netdev, queue}` couple handled. This allocation of rings guarantees a single-producer single-consumer access pattern on the kernel side, allowing faster operations. In user-space, the responsibility to make sure that no concurrent access to the same ring can occur is left to the programmer. When a packet is received by the NIC, it is first processed by an XDP program, that can choose to redirect it towards an AF_XDP socket stored into a map of type `XSKMAP`. In the initial implementation of AF_XDP, packets where first DMAed by the NIC into a kernel-owned XDP buffer, then copied into a UMEM buffer and sent to user space (a similar operation was performed for transmission). A *zero-copy* mode was later introduced [42], allowing the NIC to DMA packets directly into/from a UMEM buffer and move them to user space without expensive copies. While the

*copy* mode works on all drivers supporting XDP, the *zero-copy* mode requires an explicit driver support.

In a standard AF_XDP-based packet processing program, the NIC triggers an interrupt after the reception of one or more packets. Packets are then processed by the driver *poll* function in the NAPI software interrupt context (*softirq*) and possibly moved to the AF_XDP `rx` ring, and the application has to check the ring for the presence of new packets (a similar process happens for transmission). This can be done either with a busy loop or with the `poll()` system call that puts the program into a wait state until buffers are available. The user space application and the driver can either be executed on the same core, with the consequent cost of continuous context switching between app and *softirq*, or on separated cores, this time adding the cost of realigning the caches of the different CPU cores (cache coherency). An additional *preferred busy-polling* mode was recently introduced in [43]. With this mode, interrupts are disabled and the user space application is responsible of periodically executing the driver *poll* function with a system call (`sendto()` or `recvfrom()`). This allows running the application and the driver on a single core eliminating all the context switching and coherency traffic costs, but at the cost of executing a `syscall` for each batch of transmitted/received packets.

### 3.2.3 Packet steering mechanisms

Given the massive number of processing cores available in modern CPUs, a key problem is how to distribute traffic load among the different available CPU cores, which is important for both NFs and traditional applications. This can be achieved leveraging either software-based techniques running in the kernel, or hardware-based mechanisms available on the NIC, such as the following.

#### Receive Side Scaling (RSS) [44]

It allows a NIC to distribute incoming packets on multiple queues, which can then be processed by different CPU cores without contention (assuming no dependencies are present in the above traffic). RSS typically applies a hash function to the packet 5-tuple to identify the target queue, which guarantees that packets belonging to the same session are processed on the same core.

**Receive Packet Steering (RPS) [45]**

It is a software implementation of RSS in Linux. Whereas RSS selects the queue and hence CPU that will run the hardware interrupt handler, RPS selects the CPU to perform protocol processing above the interrupt handler. This is accomplished by placing the packet on the desired CPU's backlog queue and waking up the CPU for processing. Like in RSS, the target CPU is selected applying a hash function on the packet 5-tuple. Each receive hardware queue has an associated list of CPUs to which RPS may enqueue packets for processing.

**Receive Flow Steering (RFS) [46]**

It is an extension of RPS that redirects packets to the core where the consuming application is running. This increases the performance by improving the cache locality for data structures handling the session (both kernel and user/space processing happens on the same core), and avoids copies of the packet among cores [47].

**Ntuple filters [44]**

They define a set of rules, configured on the NIC hardware, that can (*i*) steer packets to a given queue, (*ii*) drop traffic or (*iii*) enforce specific hash options for RSS; this is called *Ethernet Flow Director* [48] on Intel NICs. Flow Director rules can be either inserted manually (*Externally Programmed Mode*) such as through `ethtool`, or automatically populated through the proprietary *Application Targeting Routing* technology (ATR).

**Application Targeting Routing (ATR) [48]**

It represents the hardware acceleration of RFS available on selected Intel NICs: the NIC driver samples some of the *outgoing* packets and automatically generates hardware rules that force the incoming traffic to be sent to the same queue/core where the application is running.

## 3.3 Benchmarking methodology

### 3.3.1 Objectives

This section defines a benchmarking methodology for the performance characterization of XDP and AF_XDP, with the final objective of determining the best technology to be used on servers that host both traditional (i.e., computing intensive) and data plane (i.e., network intensive) workloads. For this aim, we identified three classes of traffic that must be handled by our server, each one characterized by a different processing path in the Linux networking stack. **Dropped traffic** refers to packets discarded by the NF, such as in case of a firewall, a DDoS mitigator or (partly) a traffic shaper. **Pass-through traffic** refers to packets that are forwarded to a remote destination after being processed by one or more NFs on the local server, such as in case of a load balancer redirecting packets towards backends running on different servers. **Local traffic** refers to packets that have to be processed by an application running on same server as the NF, e.g., traffic that is inspected by a firewall and is terminated on a local application, such as a Kubernetes pod running locally. Even though the above three scenarios are usually combined in a common deployment, we analyzed them in isolation to facilitate the profiling of the technologies under test, and then used results to determine their best combination in real use cases.

For each class of traffic we analyzed four cases. First, we evaluated raw I/O performance (**Pure I/O**), i.e., the impact of non-avoidable components such as the NIC driver and the basic XDP and AF_XDP mechanisms on the overall throughput, using a minimal program that simply swaps the MAC addresses of the packet. This test highlights the maximum theoretical throughput obtained by each technology, which can be considered our ideal target. Second, with respect to the *dropped* and *pass-through* test cases, we profiled the performance by running software with increasing complexity, either in terms of **CPU demand** (i.e., programs whose processing logic has different degrees of complexity) and **Memory demand** (i.e., different amount of RAM addressed by the application). The former aims at determining the impact of the data plane program complexity on the overall performance, while the latter aims at determining the impact of the amount of allocated memory, which, surprisingly to us, is not orthogonal to the chosen processing technology. Processing complexity was measured by creating a program that recomputed the L4 checksum of the packet an increasing number of times. Memory demand was measured by allocating an

array with increasing size and performing, for each packet, one random access in the above memory. In this respect, we verified how the generation of a random number introduces a minimal additional (but constant) CPU processing cost and has almost no memory impact. At the same time, random memory accesses make irrelevant the hardware pre-fetching capabilities of the CPU, hence avoiding the forecast of future requests and the masking of memory access costs. Finally, we ended our evaluation with a (proof-of-concept) real application (**Traditional NF**), to confirm that our findings are actually verified in a realistic scenario.

Our analysis focuses on the throughput of the technologies under test; albeit the latency represents another important metric, for the sake of space we leave its evaluation for a future work.

### 3.3.2 Benchmarked technologies

For in-kernel packet processing we executed our NFs in the standard **XDP native mode** (**XDP** in Table 3.1) to leverage all the advantages of early packet redirection/discarding.

For user space packet processing we executed our AF_XDP-based NFs in three different modes, all relying on the *zero-copy* user-kernel interaction. In **standard mode** (**AF_XDP** in Table 3.1) the NIC driver execution is triggered by the traditional interrupt/NAPI based mechanism; we performed a busy loop to check for the presence of new descriptors in AF_XDP rings in our user-space packet processing thread. In the **system call** mode (**AF_XDP sysc** in Table 3.1) we enabled the `SO_PREFER_BUSY_POLLING` flag to trigger the execution of the NIC driver through a system call executed in a (user-space) busy loop, leaving interrupts disabled, and configured the network interface as suggested in [43][1]. Instead, the **poll mode** (**AF_XDP poll** in Table 3.1) replaced the busy loop mechanism with the `poll()` system call, leaving the user space code in a waiting state until packets are received, all triggered by an interrupt.

Enabling AF_XDP sockets changes the way packet buffers are managed and how the code of the NIC driver handles packets, therefore impacting also XDP performance. Hence, we defined two **combined test modes** (**XDP-sk** and **XDP-sk**

[1] `echo 2 | sudo tee /sys/class/net/<ifname>/napi_defer_hard_irqs`
`echo 200000 | sudo tee /sys/class/net/<ifname>/gro_flush_timeout`

| Test mode | Processing location | User space packet notification mode | Driver execution mode | AF_XDP enabled |
|---|---|---|---|---|
| XDP | Kernel | - | Interrupt | No |
| AF_XDP | User | Busy loop | Interrupt | Yes |
| AF_XDP sysc | User | Busy loop | Syscall | Yes |
| AF_XDP poll | User | poll() | Interrupt | Yes |
| XDP-sk | Kernel | poll() | Interrupt | Yes |
| XDP-sk sysc | Kernel | Busy loop | Syscall | Yes |

Table 3.1 Main characteristics of the encompassed test modes.

**sysc** in Table 3.1, the former relying on the traditional interrupt-based mechanism to trigger the NIC driver, the latter relying on a system call executed in a busy loop running in user space), in which AF_XDP sockets are enabled even if packets are completely processed at the XDP level and never reach user space.

All our tests (unless specified differently) run on a single CPU core, to prevent entering the multi-core scalability domain, whose study is left to a future work. While this configuration fits perfectly in-kernel processing, where a single processing context is scheduled (i.e., the NIC interrupt context), it may raise some concern in the user space case, in which both the interrupt and the user space application are potentially running at the same time. For example, [49] suggests to handle NIC interrupt requests (IRQ) and applications (hence, kernel vs user-space processing) on different CPU cores to avoid context switches and maximize performance.[2] Nonetheless, we scheduled them on the same core to simplify the comparison with other technologies, paying attention not to affect the validity of our results. In fact, we always tuned the offered load to maximize the throughput of the system, achieving a result that was more than half the one obtained with two distinct CPU cores. This guarantees the fairness of our results, because our tuning avoids the livelock phenomenon in which the code running at the highest priority, i.e., the kernel, consumes most of the resources while the rest tends to starve, as shown in [50].

[2]Notably, this does not apply to the *AF_XDP sysc* case in which IRQs and application *have* to be scheduled on the same core to achieve decent performance; more in Section 3.2.2.

### 3.3.3 Testbed

Our testbed is composed of two servers equipped with a dual-port Intel XL710 40 Gbps NIC, only one port used, connected back-to-back. One server operates as Device Under Test (DUT) and the other one as tester/load generator. Both machines feature an Intel Xeon Gold 5120 14-cores CPU @ 2.20 GHz with Hyper-Threading and Turbo Boost disabled. The processor is provided with 32 KiB of per-core L1 data cache (corresponding to 512 x 64B lines), 1 MiB of per-core L2 cache ($\sim$16K x 64B lines) and a 19.25 MiB unified L3 cache ($\sim$315K x 64B lines). The servers run Ubuntu 20.04.4 LTS with kernel 5.14. The traffic is terminated on the DUT for the *Dropped* and *Local* tests, while is sent back to the traffic generator for the *Pass-through* tests. The DUT supports Intel DDIO technology, that allows the network card to DMA packets directly to/from the Last Level Cache (LLC, a.k.a. L3), hence avoiding high latencies due to the access to the main memory. As suggested in [51] (and confirmed in our tests), we increased the number of LLC ways available to Intel DDIO from 2 to 6[3] to improve the throughput of NFs and reduce packet losses.

For the dropped and pass-through test cases we used MoonGen as packet generator, generating minimum size UDP packets (64B Ethernet frames) towards the XDP/AF_XDP NF running on the DUT. We measured the throughput of the NFs according to RFC2544, tuning the input rate till the packet loss was lower than 0.1%. For local traffic tests we selected `memcached` [52] as a sample application, since it is likely to be deployed at the edge of the network and it is rather network intensive. We executed it with a variable number of threads and with a memory limit extended to 128MB (default is 64MB), and we pinned it to a set of cores (`taskset` command), either shared or disjoint from the ones used by the NF depending on the benchmarked technology (more details in Section 3.6). Requests on the tester machines were generated with Memoslap[4] running on four cores, 128 clients per core, each one establishing a TCP connection and requesting items with random keys for ten seconds. The `sar` tool was used to measure the CPU utilization of the DUT, split between user space (*User*), system calls (*System*) and software interrupt processing (*SoftIRQ*), the latter two both related to kernel space code. We also leveraged the `perf` tool to monitor the number of LLC hits and misses (the latter representing the number of memory accesses), since (*i*) memory access latency is

[3] `sudo wrmsr 0xc8b 0x7e0`.

[4] https://github.com/FedeParola/memoslap

one of the main bottlenecks in packet processing [53] and (*ii*) the LLC is the target of packet transfers to/from the NIC (DDIO). Due to space concerns, the above numbers are presented only when they provide some insights on the causes of the achieved throughput. In all cases we repeated our measurements 10 times and our plots show the average value and the standard deviation as error bars.

All the code used for testing is publicly available.[5]

## 3.4 Dropping traffic

### 3.4.1 Pure I/O performance

Results in Figure 3.2 show the performance in terms of dropped traffic of the technologies under test in a pure I/O scenario (packets are only touched by swapping their MAC addresses). In general, XDP packet dropping is highly efficient, as it avoids additional kernel processing (if the Linux network stack is traversed) or to exchange frames on AF_XDP rings (if AF_XDP processing is involved). However, XDP dropping performance are even better if AF_XDP sockets are enabled, with a 21% improvement when the driver is executed in interrupt mode (*XDP-sk*). Numbers in Figure 3.3, which shows LLC accesses (hits and misses), seem to suggest a higher memory usage of pure XDP, which needs one LLC load and one LLC store per packet, while other technologies do not need the store operation. However, this LLC store is due to a `prefetchw()` operation in the NIC driver, which prepares the memory area to store the metadata for packet transmission (the `xdp_frame`). However, since the `prefetchw()` assembly instruction is executed asynchronously, we detected no difference in performance when removing the above operation from the driver, even when this memory region is not needed (i.e., when dropping traffic).

We speculate that the root cause of the performance improvement in *XDP-sk* is the different buffer management model introduced by AF_XDP, since, to the best of our knowledge, this is the only main difference when enabling AF_XDP sockets in the drop scenario. In addition, Figure 3.4 shows the execution context of the packet processing code, namely *user space*, *system call* or *software interrupt*, where the last two are both in kernel space. Interestingly, the usage of a system call to retrieve

[5] https://github.com/FedeParola/xsknf

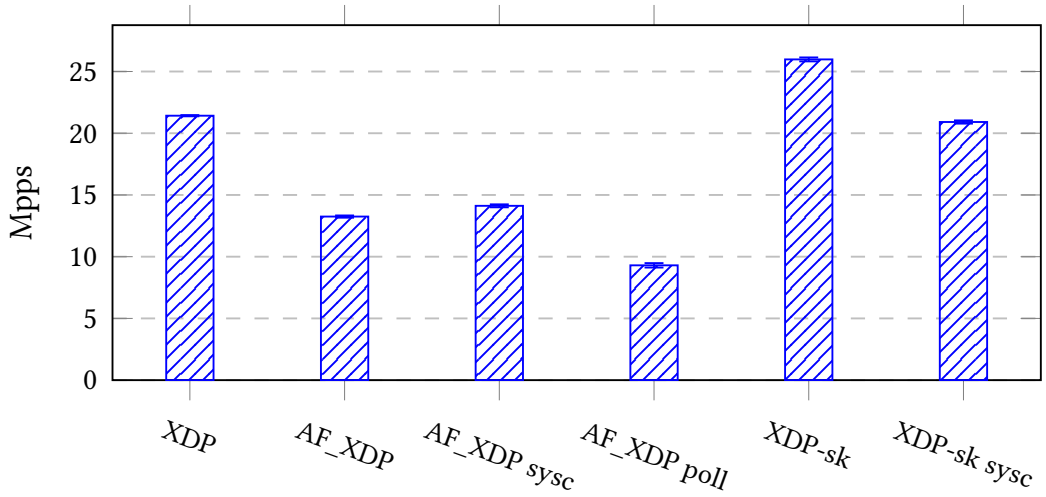


Fig. 3.2 Maximum manageable rate in the pure I/O test case (mac address swap) when dropping packets.

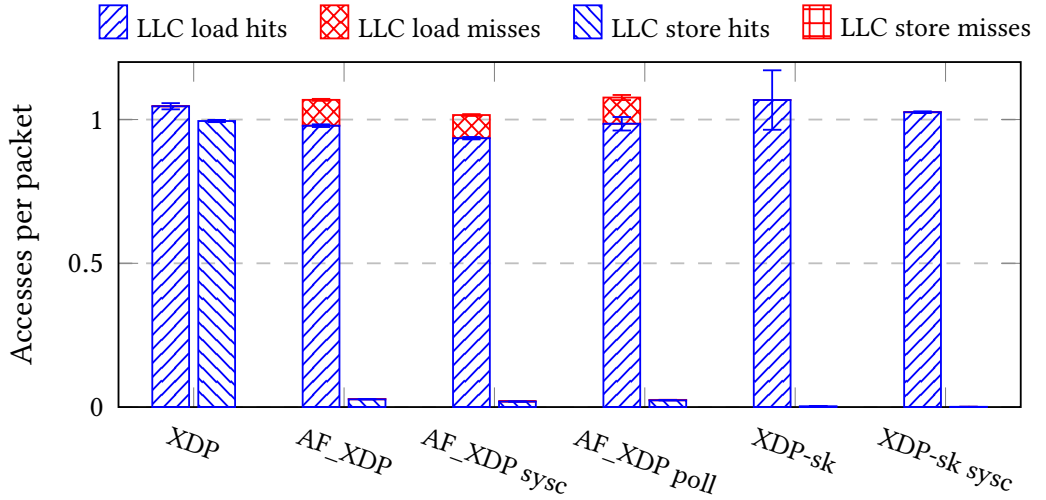


Fig. 3.3 Per-packet LLC accesses in the pure I/O test case (mac address swap) when dropping packets.

packets is convenient when dropping traffic in user space (*AF_XDP sysc*), but it has a negative effect when this operation is performed in the kernel (*XDP-sk sysc*), since we still have to spend some processing time in user space just to trigger another driver loop. Finally, Figure 3.4 shows also that technologies that include a context switching (e.g., part of the processing is done in user space, part in SoftIRQ) tend to perform worse, suggesting the opportunity to choose a technology that completes all the processing in the same context.

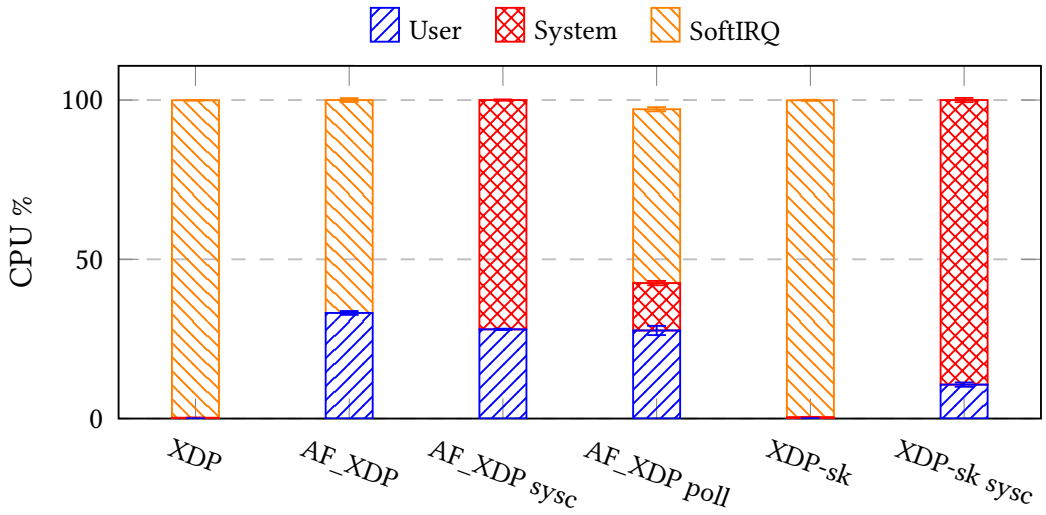


Fig. 3.4 CPU usage in the pure I/O test case (mac address swap) when dropping packets.

In conclusion, dropping packets at the kernel level always proved to be more efficient (*XDP*, *XDP-sk* and *XDP-sk sysc*), with an extra improvement when enabling AF_XDP sockets (*XDP-sk*), suggesting a more effective buffer management model introduced by AF_XDP.

### 3.4.2 Impact of memory demand

Figure 3.5 shows the throughput achieved by the most effective I/O configurations (namely *XDP*, *AF_XDP-sysc*, and *XDP-sk*) while increasing the memory allocated to our NF and randomly accessed. Results show a stable throughput as long as the allocated memory is limited in size, with an advantage of in-kernel (*XDP* and *XDP-sk*) over user space (*AF_XDP*). However, the gap shrinks when the amount of allocated memory exceeds 16K cache lines (the size of our L2 cache), with user space processing eventually outperforming *XDP* first and *XDP-sk* next.

Comparing these results with the number of per-packet accesses in the LLC cache (Figure 3.6) we can see that they remain stable as long as the size of the allocated memory fits in the L1/L2 caches. All LLC accesses in this region are related to a load/store in the packet buffer: in fact, DDIO transfers (through DMA) packets to/from the LLC, hence all packet accesses result in an LLC access. As in the pure I/O test case, pure XDP presents one additional LLC access due to a *prefetch* operation. While this operation should intuitively affect the memory scalability, we

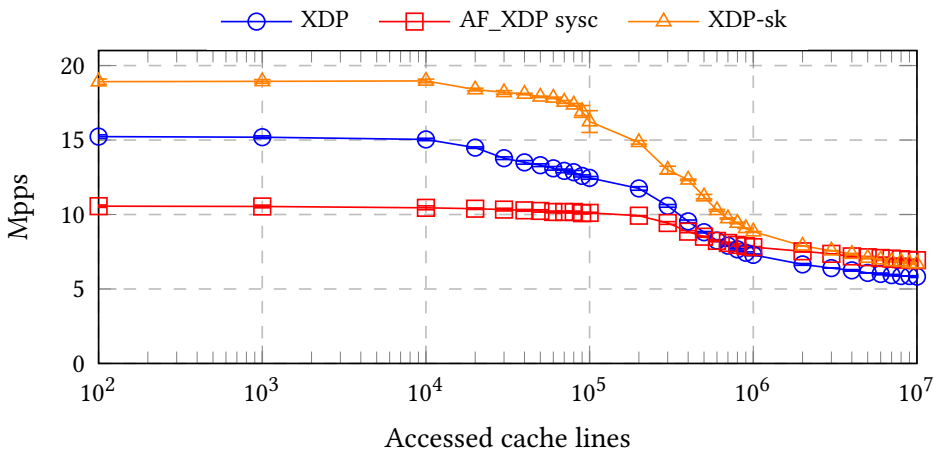


Fig. 3.5 Impact of an increasing memory demand on the throughput when dropping traffic.

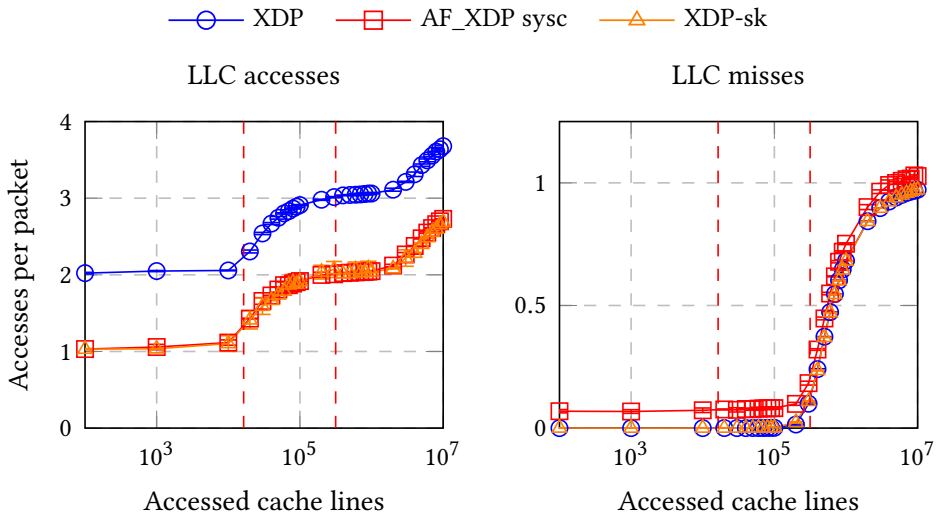


Fig. 3.6 Number of accesses in the LLC cache (reads + stores) per packet, with increasing amount of allocated memory, when dropping traffic. The vertical dashed lines represent the size of the L2 and L3 caches. The line for *XDP-sk* is totally overlapped with *AF_XDP-sysc* in *LLC accesses* and with *XDP* in *LLC misses*.

did not detect any performance difference when removing that operation. In general, pure XDP performance showed to be the most affected by memory demand, with an improvement achieved when enabling AF_XDP sockets (*XDP-sk*). However, user space processing with AF_XDP (*AF_XDP-sysc*) proved to be the most resilient solution with respect to memory demand, which suggests the presence of some important difference beyond the buffer management model, whose identification is left for future studies. For instance, this is confirmed in Figure 3.8 (e.g., 5M entries),

in which *AF_XDP-sysc* is less efficient when a limited memory is involved, but it outperforms *XDP-sk* when a large amount of memory is requested (in that case, the identifiers of many TCP sessions).

In general, processing packets in user space with AF_XDP proved to be less affected by the amount of memory accessed by the NF.

### 3.4.3 Impact of CPU demand

Figure 3.7 shows the throughput achieved by the technologies under test with NFs of increasing processing complexity. Unlike the memory case, all technologies were similarly affected by this increase in CPU requirements, with in-kernel processing keeping its performance gap over user space (even if this became less relevant with higher processing complexity becoming dominant over the I/O processing cost). As expected, we did not detect variations in the number of LLC/memory accesses, that always remained equal to the ones measured in the pure I/O test. Interestingly in this test we did not experience the gap between the *XDP* and the *XDP-sk* modes, likely because the NF processing cost dominates over the cost of pure I/O.

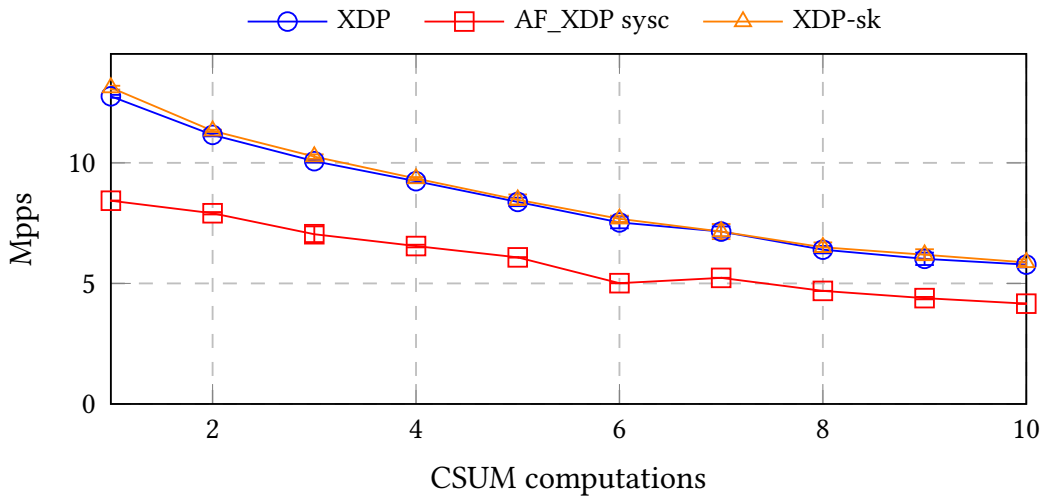


Fig. 3.7 Impact of an increasing CPU demand on the throughput when dropping traffic.

### 3.4.4 Traditional NF performance

In this experiment we evaluated the performance of a realistic NF, namely a simple L4 firewall dropping all traffic whose 5-tuple matches a set of pre-configured rules stored in a hash table. The main parameter that influences its performance is the number of ACL entries, as well as the number of different flows matching those entries. Therefore, in our test we scaled this variable, influencing the degree of memory dependency of the function.

We wrote two versions of the firewall, an XDP-based and an AF_XDP-based one, keeping them as similar as possible. To avoid a bias in the results due to the characteristics of the hash table, the AF_XDP firewall leveraged a user space clone of the eBPF hash map available in kernel,[6] which differs only in how concurrent read and write operations are handled. The eBPF hash map leverages Read-Copy-Update [54], which supports multiple reads to be executed in parallel with one update, with a negligible cost on the read side. Unfortunately there is no ready-to-use implementation of this mechanism in user space, so we decided not to support this type of concurrency in our map and our tests included only *read* operations, in which the overhead would be negligible anyway; hence, we pre-populated the maps before starting our measurements.

We configured the firewall with an increasing number of ACL entries (each one with a different source IP address), and tested the maximum throughput achievable by the NF when dropping all the received packets, which were randomly distributed across all configured sessions. Results in Figure 3.8 confirm the advantage of dropping packets at XDP level (both with AF_XDP sockets enabled and with standalone XDP) but only as long as the size of the ACL is limited (and fits the L2 cache size size). *XDP* reaches up to 29% higher throughput compared to *AF_XDP sysc*, but this advantage tends to shrink as the number of sessions increases (i.e., when the cost of memory access tends to dominate over pure I/O cost). When the size of the ACL exceeds 10K entries, there is no appreciable difference between the two technologies,

[6] For the sake of precision, the data structure has the following features: it uses the `list_nulls` double linked list of the kernel to store elements that collide on the same bucket; the maximum size of the map is defined at initialization time and the number of buckets is defined rounding this number to the next power of two; the memory for map nodes (storing both the key and the value) is pre-allocated in a contiguous area; the `jhash` function is used for hashing and the hash value is mapped to the corresponding bucket selecting its lower $n$ bits (with $2^n$ buckets); when an element is modified (added, deleted or updated), the concurrent access to the corresponding bucket is protected with a spin lock.

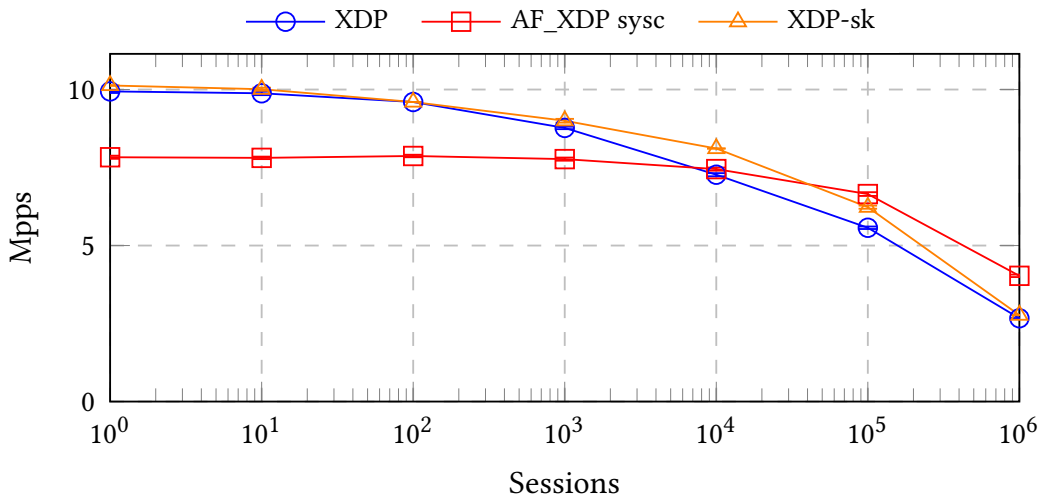


Fig. 3.8 Firewall throughput with an increasing number of processed sessions.

and for an even larger number of sessions, dropping packets in user space becomes much more efficient with a 51% performance improvement over XDP. This confirms the better memory efficiency of AF_XDP found in our *Memory demand* test: while a hash map makes hard to predict in advance when the data structure starts generating accesses in the LLC, our cache measurements (omitted for the sake of brevity) show an increase of LLC accesses in the 10K-100K sessions range.

**Takeway 1:** Dropping packets at the XDP level (*XDP* and *XDP-sk*) is more efficient when the packet processing function operates mainly in the lower levels of cache (L1 and L2), thanks to the smaller amount of code traversed to process the traffic. Bringing packets in user space (*AF_XDP sysc*) results advantageous when the NF becomes more memory bound, hence representing the most effective solution for memory intensive processing logic.

## 3.5 Pass-through traffic

### 3.5.1 Pure I/O performance

We evaluated raw packet I/O performance by measuring the maximum throughput when performing a swap of MAC addresses and sending back the packet out of the

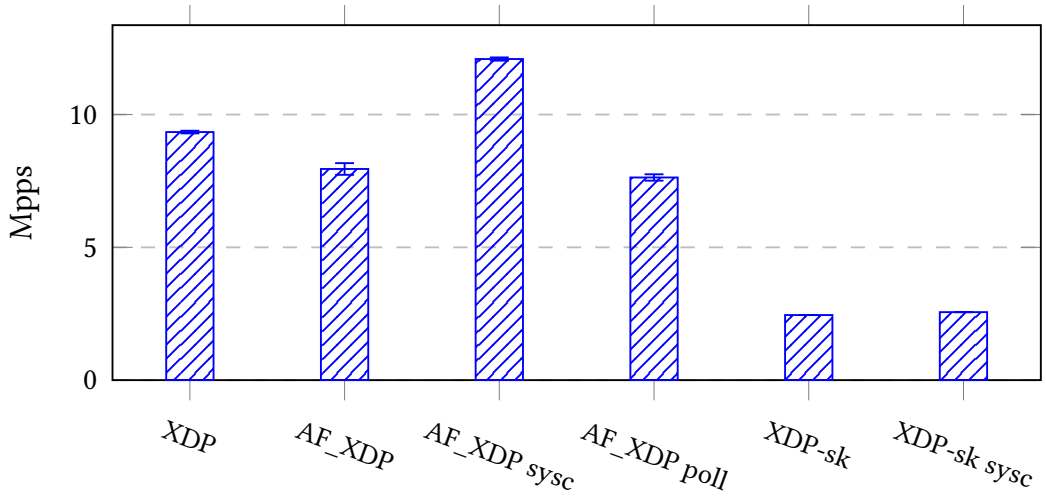


Fig. 3.9 Maximum manageable rate in the pure I/O test case (mac address swap) when redirecting packets out of the receiving interface.

receiving interface (return code `XDP_TX` at the XDP level).[7] Figure 3.9 shows that the clear winner is *AF_XDP sysc*, with a 29% improvement over XDP. With respect to user space processing, *AF_XDP sysc* outperforms *AF_XDP*, hence differing from the dropping test case in which interrupt-based or system call-based modes brought to similar performance. Instead, *AF_XDP* yielded a lower throughput, which can be explained with the need to periodically perform a system call to inform the driver of the presence of new packets to transmit. This represents an operation that consumes a CPU time similar to the (less efficient) `poll()` based mode, as shown in Figure 3.11 (*AF_XDP* and *AF_XDP poll* bars). Curiously, when redirecting packets with *XDP-sk**, we always obtained a very low throughput (about 2.5 Mpps). The reason may be due to the number of LLC accesses shown in Figure 3.10: while the number of LLC *loads* per packet is similar for all tests (and in line with the results of the drop test case), a notable difference exists for *store* operations. This number ranges from *0.1* per packet (with *AF_XDP*) to *one* per packet (with *XDP*) (unlike the drop test case, here the additional LLC write is needed to populate the `xdp_frame` metadata for transmission; see Section 3.4.1) till *two* per packet (with *XDP-sk**). In fact, an analysis of the Linux kernel (at least till version 5.14) shows that when AF_XDP sockets are enabled and packets are re-transmitted at the XDP level, the packet is

[7] This chapter does not include the results when the traffic is redirected on a different interface, which can be achieved with return code `XDP_REDIRECT` at the XDP level, and with minor changes at the AF_XDP level. In the above conditions, our experiments showed lower performance for both XDP and AF_XDP, with a trend that is very similar to the presented one.

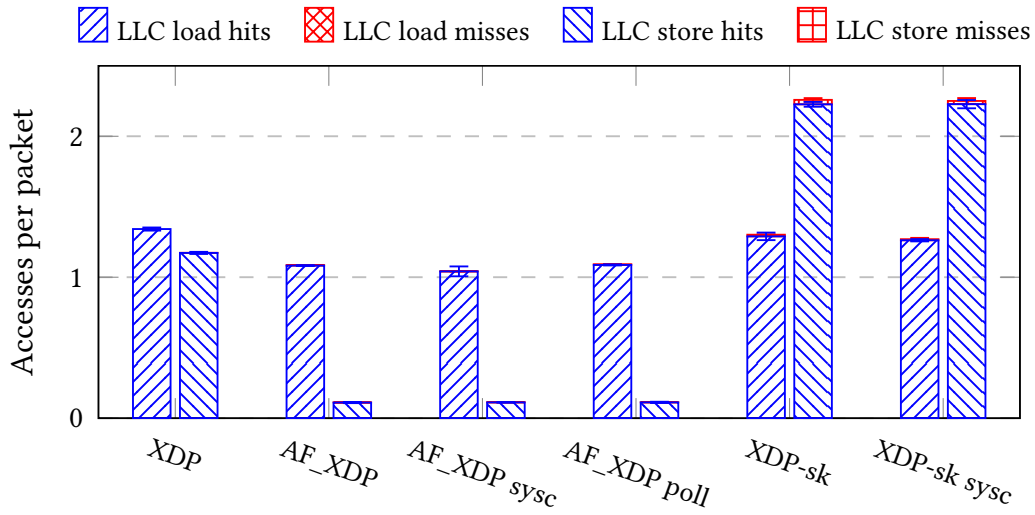


Fig. 3.10 Per-packet LLC accesses in the pure I/O test case (mac address swap) when redirecting packets out of the receiving interface. Misses can hardly be seen since they are close to zero.

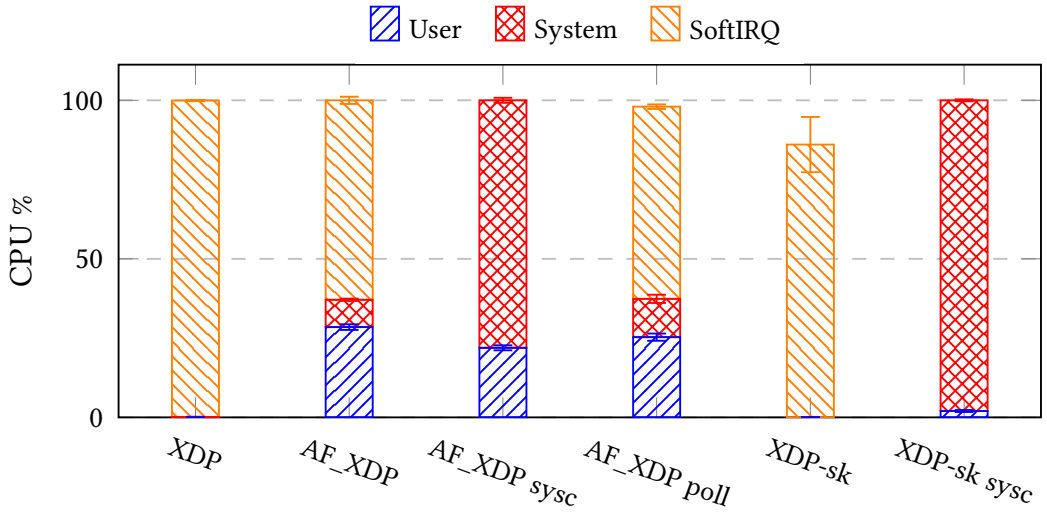


Fig. 3.11 CPU usage in the pure I/O test case (mac address swap) when redirecting packets out of the receiving interface.

copied from its UMEM frame to an in-kernel XDP page before being sent out of the interface. This expensive operation is probably the cause of the higher number of LLC stores and consequent lower throughput of the *XDP-sk* test cases.

### 3.5.2 Impact of memory demand

Results of the memory test shown in Figure 3.12 and Figure 3.13 underline a trend similar to the one observed in the dropping traffic scenario, with AF_XDP broadening its gap over XDP as memory accesses shift from L1/L2 caches to the LLC and to main memory. This gap goes from a 19% improvement when the function uses little memory to a maximum of 39% higher throughput when a notable number of accesses hits the LLC and the main memory.

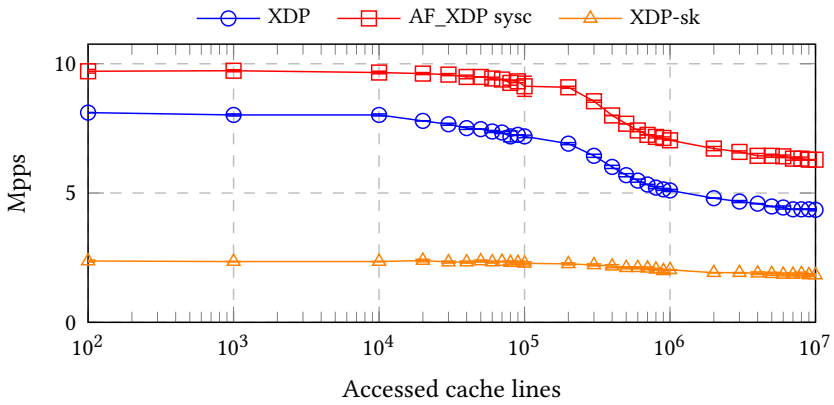


Fig. 3.12 Impact of an increasing memory demand on the throughput when redirecting traffic.

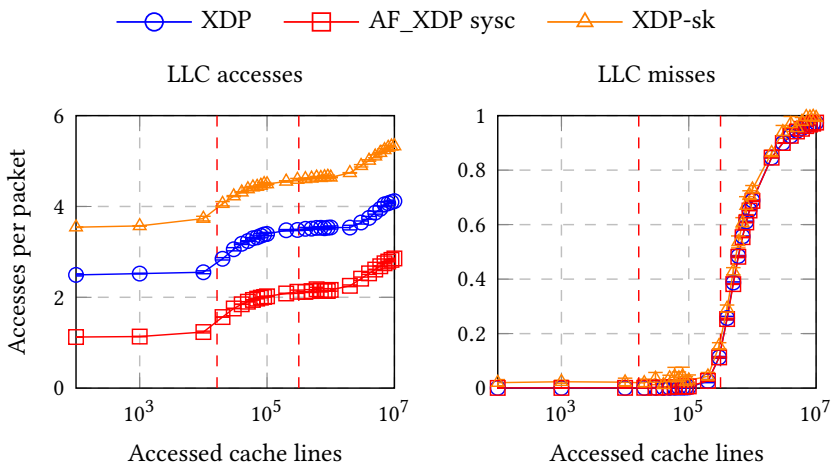


Fig. 3.13 Number of accesses in the LLC cache (reads + stores) per packet, with increasing amount of allocated memory, when dropping traffic. The vertical dashed lines represent the size of the L2 and L3 caches. LLC misses are almost overlapped.

### 3.5.3 Impact of CPU demand

Similar considerations apply to the CPU-intensive test (Figure 3.14); *XDP* and *AF_XDP sysc* score similar, with the limited advantage of the latter that disappears when increasing the complexity of the processed function, i.e., when the raw I/O cost becomes less relevant. The reduced memory efficiency of *XDP-sk* has a huge impact in this CPU-intensive test as well, hence sitting at the bottom.

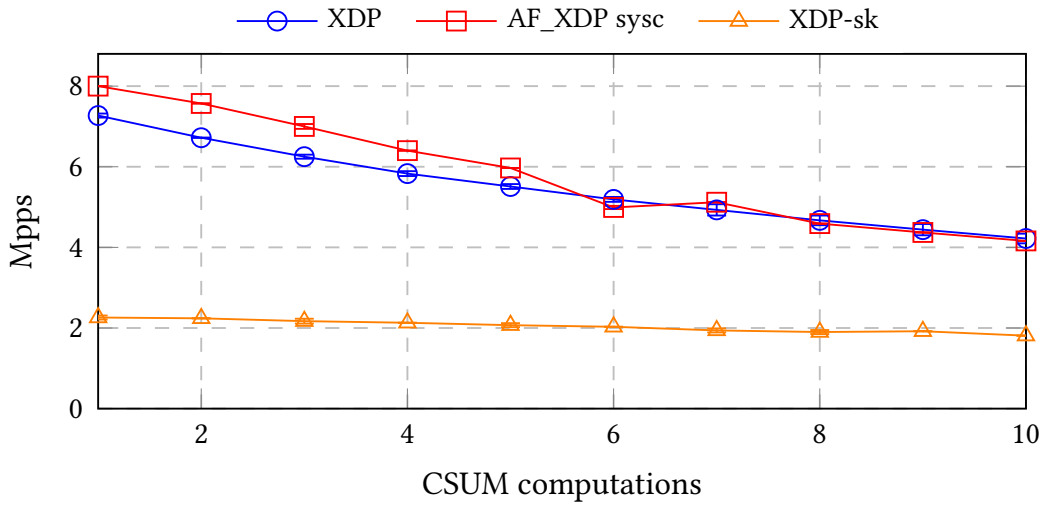


Fig. 3.14 Impact of an increasing CPU demand on the throughput when redirecting traffic.

### 3.5.4 Traditional NF Performance

To assess the performance of traffic redirection with a realistic function, we wrote a minimal load balancer that parses the packet till level 4 and uses the 5-tuple to access an hash map of the active sessions. If the lookup is successful, the retrieved value contains the information to update the packet, that can be either the destination IP, port and MAC address of the backend for incoming packets (client to service), or the original service IP and port for return packets (service to client). In case the lookup fails, a new load balancing decision is taken and two entries are added to the table of active sessions to handle both incoming and return packets. In the end, the fields of the packet are updated and the packet is sent back on the receiving interface. As presented in Section 3.4.4 with respect to the user space implementation of the hash map, we limited our performance measurements on packet processing

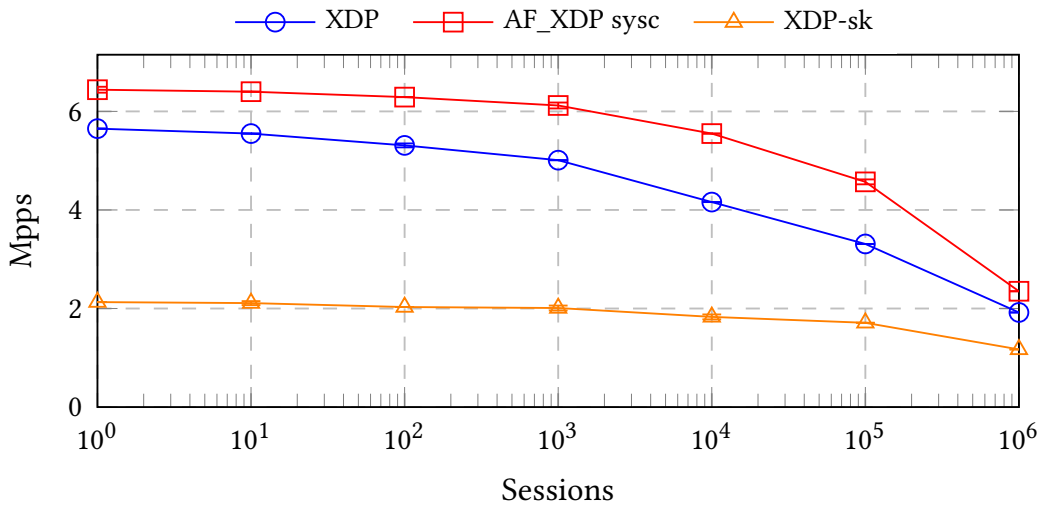


Fig. 3.15 Load balancer throughput with an increasing number of active sessions.

scenarios involving only lookup operations, avoiding *write* operations. Therefore, we populated in advance the table of active TCP sessions of the load balancer generating traffic that is randomly distributed among those sessions.

Figure 3.15 shows the maximum throughput handled by the NF for an increasing number of active sessions. For a small number of sessions (i.e., limited memory used) the performance advantage of AF_XDP against XDP is reduced compared to the one measured with raw I/O performance (a 14% improvement vs the 29% recorded in the former test). However, when the number of sessions increases and the NF becomes more memory bound, the same behavior observed in the dropping scenario applies, with user space processing increasing its performance gap over XDP, with a maximum of 38% higher throughput when processing packets of 100K different sessions.

**Takeway 2:** *AF_XDP sysc*, i.e., AF_XDP sockets with system call-triggered driver, provides the highest performance when processing pass-through traffic. The advantage over XDP is limited for simple packet processing functions but increases as the logic becomes more memory bound, requiring frequent accesses to the LLC and to the main memory. In general, considering the higher flexibility of user space processing that, unlike eBPF, does not impose any limitation (e.g., on the program size, on the type of data structures used, on loops, etc.), AF_XDP sockets should be preferred for pass-through traffic with respect to in-kernel processing.

## 3.6 Local traffic

This section investigates the case of traffic processed by one or more NFs before landing on an application running on the local server, usually as a container (e.g., a Kubernetes pod). In this case, the traffic has to traverse the entire TCP/IP stack before data is eventually delivered to the application. In our analysis we do not consider the case of applications running in a virtual machine because of the increasingly diffusion of cloud-native workloads, particularly with respect non-NF applications running in telco-oriented datacenters.

While processing a packet at XDP level, the `XDP_PASS` return code can be used to inform the kernel that the packet has to continue its journey in the standard network stack and reach the application running locally. Instead, for what concerns AF_XDP, traffic is handled in user-space and therefore we need a way to re-inject packets into the kernel, which is needed to complete the remaining TCP/IP processing and deliver the packet to the application.[8] Therefore we leveraged a `veth` interface to re-inject the packet in the kernel; however, this introduces more overhead due (*i*) to the necessity to perform an additional copy of the packet, as `veth` devices do not support the *zero-copy* mode,[9] and (*ii*) to the additional context switch. Therefore, given that the *zero-copy* capability is a property of the UMEM, we relied on two different UMEMs, the first one bound to the physical interface and operating in *zero-copy*, while the second one associated to the `veth` and operating in *copy* mode. This solution requires an additional copy of the packet between the two UMEMs each time a packet traverses the two interfaces, resulting in *two* copies per packet, the first in user space (with AF_XDP) and the second in the kernel (on the `veth`). Unfortunately, leveraging a single UMEM shared among the two interfaces does not solve the problem. In fact, while the first (user-space) copy would be avoided, this method requires the physical interface to work in *copy* mode, resulting anyway in two copies per packet for local traffic (performed by the kernel on each one of the two interfaces), but losing the performance advantage of *zero-copy* on the physical interface in case of pass-through traffic.

[8] We did not encompass user space implementations of the TCP/IP stack (e.g., [17]) since they are not widespread and require patched applications in order to be leveraged.

[9] At the time of writing, all physical and virtual NICs supporting XDP are also compatible with AF_XDP sockets, but only physical NICs support the more efficient *zero-copy* mode, even if an attempt to add it to the `veth` driver was made in the past [55].

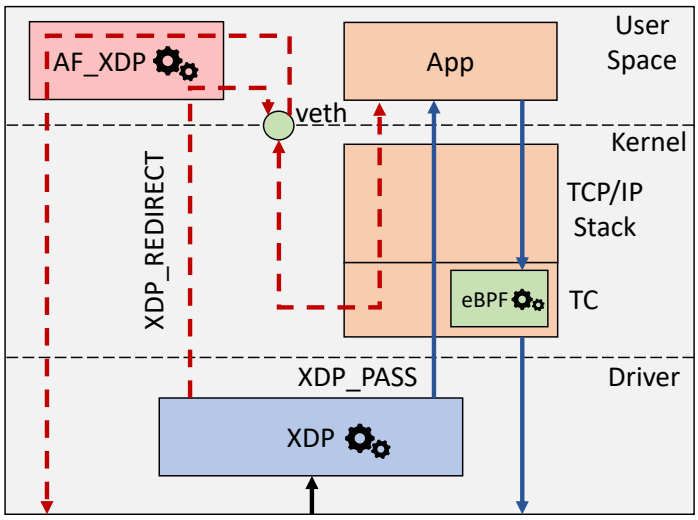


Fig. 3.16 Path of traffic reaching a local application, when the NF runs as XDP code (blue continuous line) or as AF_XDP code (red dashed line).

While with AF_XDP the traffic is processed on the ingress and the egress paths (thanks to the packet redirection in user-space done by the `veth`), XDP programs can only be attached to ingress hooks. In this case, egress traffic processing leverages the *TC* eBPF hook, which executes the same packet processing function on traffic leaving the server. Figure 3.16 highlights the different paths followed by the traffic to reach a local application, depending on where the NF is implemented (XDP or AF_XDP).

### 3.6.1 Pure I/O performance

The first test determines the raw I/O performance limited to the early part of the Linux TCP/IP stack, by assessing the overhead of adding custom NF processing to packets reaching the local application. Particularly, it assesses the cost of the initial processing of the packet *before* reaching the TCP/IP stack, which is different in XDP and AF_XDP + `veth` cases, while the remaining processing stack is the same for both. We used Moongen to generate UDP traffic with non-existing MAC addresses, and executed an XDP/AF_XFP program that performs a MAC swap on the packets, which are then passed to the network stack. Due to the wrong destination MAC address, which does not correspond to the ones present on the server, packets are

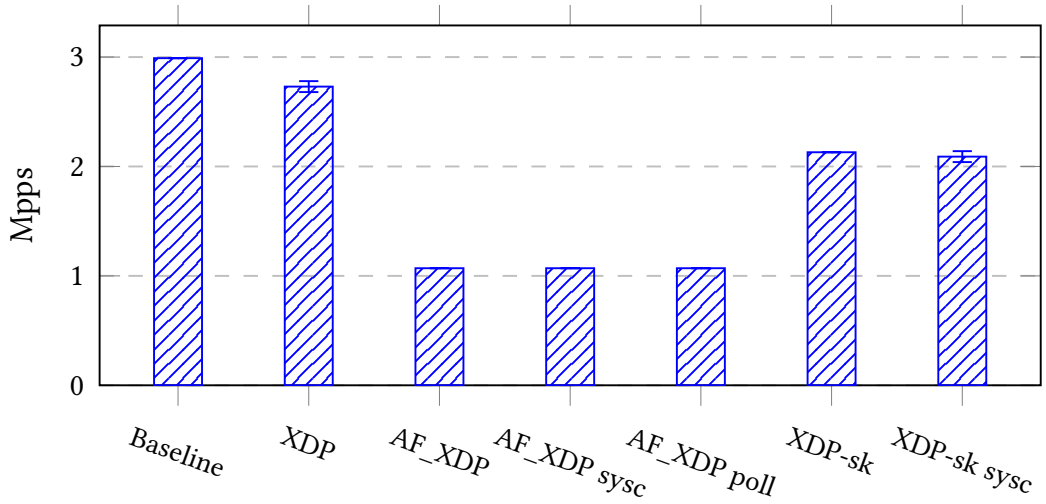


Fig. 3.17 Throughput when delivering a packet to the TCP/IP stack based on the different processing paths highlighted in Figure 3.16.

dropped very early in the network stack. Results in Figure 3.17 show that adding some processing at the XDP level has very little overhead (9%) with respect to the baseline (i.e., when packets are dropped by the kernel without being first processed by XDP/AF_XDP). On the other hand, when moving packets into user space and then back into the kernel, the hit on performance is considerable, with a 64% reduction of throughput. Interestingly we did not experience any difference for the three different AF_XDP-based modes.

On the other side, *XDP-sk** experiments measure the cost of enabling AF_XDP sockets on traffic that is not redirected in user space. This is the price we may have to pay if we want to leverage both XDP and AF_XDP at the same time, e.g., by splitting the traffic and handling it in part with AF_XDP (which represents the best choice for pass-through traffic, Section 3.5), in part with XDP (which represents the best choice for dropped traffic, Section 3.4). One of the reasons of this additional cost is due to the necessity to copy the received packet from the UMEM to a new buffer before sending it up to the network stack, assuming the NIC operates in the more efficient *zero-copy* mode. This is needed because the UMEM can be modified in user space, hence the packet could be corrupted during kernel processing, causing unexpected behaviors. Vice versa, in vanilla XDP the buffer is only accessible by the kernel, hence allowing true *zero-copy* operations [56]. This cost could be prevented by using AF_XDP sockets in *copy* mode that, like XDP, receives packets in a private

kernel buffer and then copies them to the UMEM. This however would result in very poor performance for the traffic reaching the user space (in our experiments we were able to achieve a maximum of 1.17 Mpps when redirecting packets in *copy* mode, way below the performance of other *zero-copy* versions of AF_XDP shown in Figure 3.9), hence forcing us to discard this alternative. Results show that, at least in this I/O-intensive test, the overhead of enabling AF_XDP sockets is significant, with a 22% performance reduction of *XDP-sk* against pure XDP (Figure 3.17).

The second test analyzes the raw I/O performance including the entire Linux TCP/IP stack and the receiving application, by using our sample application (`memcached`), hence evaluating the impact of added packet processing in a real scenario. In this test, the number of CPU cores to be used, and the allocation of different processing tasks to each core proved to be more problematic than in the previous cases. For instance, the Linux scheduler gets confused by the polling-based working mode of AF_XDP, which looks like a user-space process always requiring more resources. Hence, the CPU allocation is partitioned among all the requesting processes (in this case AF_XDP and `memcached`), reaching the best equilibrium when the core is equally shared among the two (50% each). However, this would achieve sub-optimal performance, because the 50% allocated to AF_XDP is only partly spent in doing actual processing, while the rest is spent in empty busy polling iterations. To overcome the above limitation, in this test we used multiple cores. We dedicated one core to packet processing and then we added a number of cores to `memcached` that enabled to reach stable performance, which implies a saturation the NF core. This lead to the usage of (1+3) cores, achieving an average 96% usage of `memcached` cores, indicating that we were wasting almost no resources. It is important to notice that with this configuration the ingress TCP/IP stack is executed on the NF core (where the software interrupt of the `veth` interface is triggered), while the egress stack is executed on the application cores (where the system call sending packets is executed). In fact, when we leveraged Receive Packet Steering (Section 3.2.3) to move the network stack processing to the `memcached` cores, we simply increased load imbalance (i.e., some fully utilized cores along others with more idle time), resulting in lower overall throughput, hence confirming that the previous configuration was the best choice in our operating conditions.

For the *XDP-sk sysc* experiments, even if packets are processed at the XDP level, the code is executed in the context of the user space application, within a busy loop triggering the driver with a system call. From the Linux scheduler perspective, the

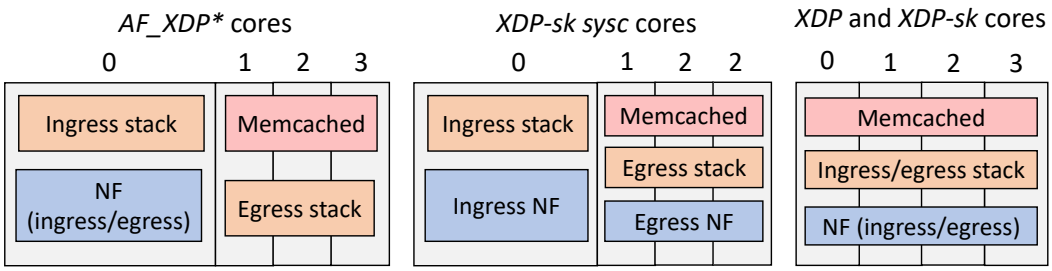


Fig. 3.18 Distribution of the cores between NF, Linux network stack and application.

NF is therefore seen as an application requiring all resources available on the core, and is subject to the same considerations that apply to AF_XDP, hence we handled it with the same (1+3) cores partitioning. For what concerns other solutions based on in-kernel, interrupt-based processing (i.e., *XDP* and *XDP-sk*), we were able to partition available cores in a more granular way thanks to RSS, allowing the NF and `memcached` to share each one of the 4 available cores according to their needs and the overall traffic load. The final cores distribution used in our test is shown in Figure 3.18.

With XDP, we achieved the highest performance by enabling *Application Targeting Routing* (Section 3.2.3), that moves the XDP processing on the same core where the recipient application is running (Figure 3.19, *XDP ATR on*). This technology is not available when processing traffic in user space because it requires the execution of a part of the driver that is bypassed by AF_XDP. Since not all network cards support ATR (or equivalent technologies), and it might not always be effective (e.g., in the case of connectionless traffic), we evaluated the performance of in-kernel processing also with ATR off (i.e., relying on the classic RSS packet steering). Figure 3.19 shows the throughput we achieved in terms of requests per second handled by `memcached`. The gap between in-kernel and user space packet processing reduced significantly with respect to the former test case (Section 3.6.1), but running our NF at the XDP level still guaranteed a considerable lead over AF_XDP, with a throughput advantage of 42% with ATR on, and 27% when this technology is not available. In this test, the performance reduction when enabling AF_XDP sockets (i.e., *XDP-sk*) is limited to no more than 5%.

Final remark, no NIC-based accelerations were used with *XDP-sk sysc* and all *AF_XDP*-based technologies, as the entire traffic goes on the single core that executes the NF. Instead, *XDP* and *XDP-sk* look more efficient when ATR is *on*, compared to

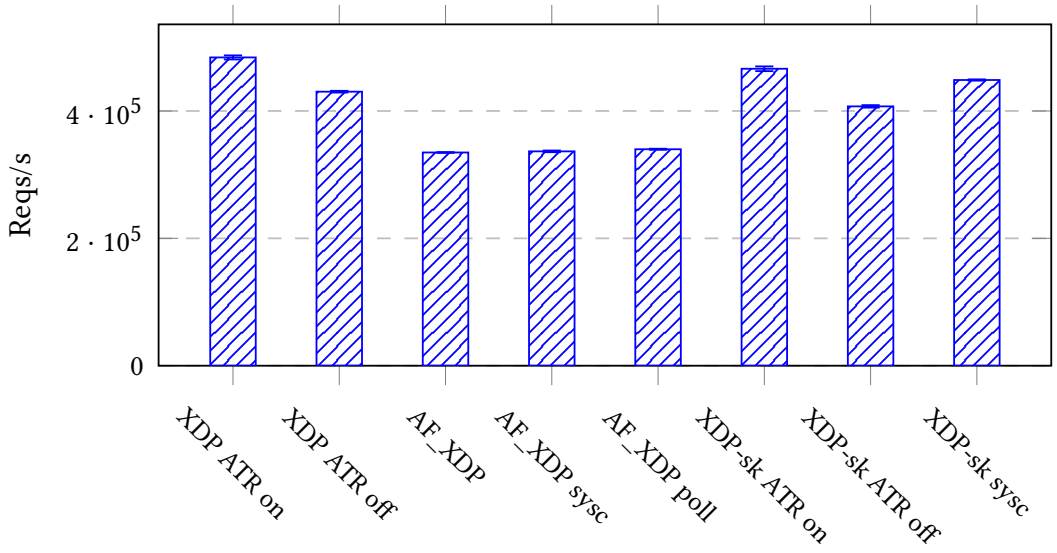


Fig. 3.19 TCP/IP-based application throughput (`memcached`) when applying some simple processing (mac swap) to packets.

the simpler RSS (active when ATR is *off*). In the XDP context, enabling RFS did not prove to have any positive impact with respect to plain RSS. This is due to the fact that RFS is applied at a later stage in the networking stack with respect to XDP, after the NIC driver has completed its operations, resulting in the XDP program and the application thread potentially being executed on different cores.

### 3.6.2 Traditional NF performance

To evaluate the impact of a more complex NF, we extended the load balancer presented in Section 3.5.4 to balance sessions also toward a local `memcached` backend running on the same server. Results in Figure 3.20 are very similar to the ones that we observed in the simple processing scenario (Figure 3.19), indicating that the cost of the NF is negligible compared to the complexity of the network stack and the application. However, our tests show that in the *AF_XDP** and *XDP-sk** cases the additional pressure put on the (single) NF core exacerbates the load imbalance problem caused by the rigid partitioning of cores. In fact, in this test the bottleneck imposed by the NF core caused our application cores to be leveraged only at about 88% (compared to the 96% of the mac swap test), slightly increasing the gap between in-kernel and user space processing performance.

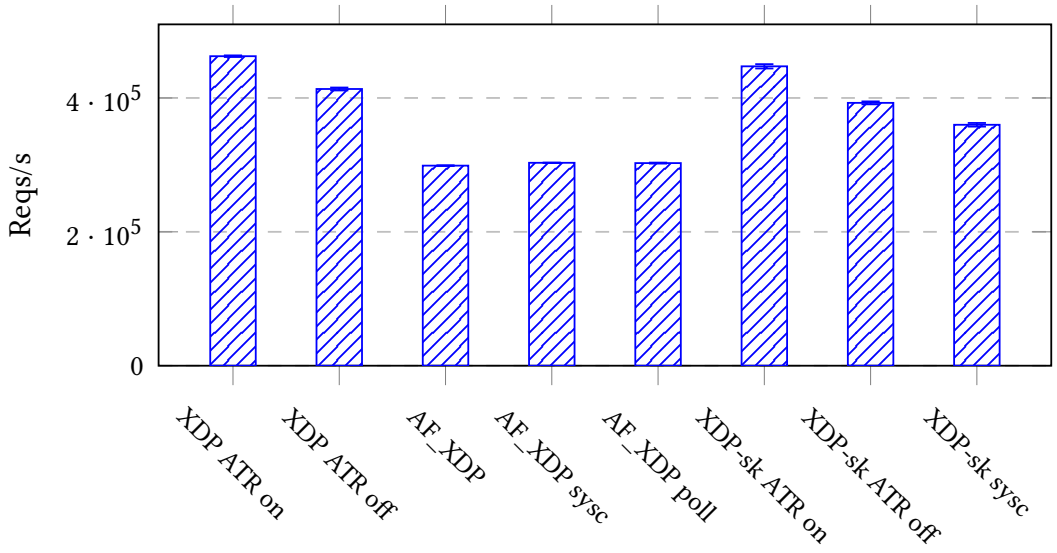


Fig. 3.20 TCP/IP-based application throughput (`memcached`) when applying some complex processing (load balancing) to packets.

**Takeway 3:** When processing traffic that is directed to a service running on the local server and leveraging the TCP/IP stack, XDP is much more efficient than AF_XDP, thanks to its smooth integration with all the TCP/IP processing code that runs also in the kernel, hence avoiding expensive kernel-to-user (and vice versa) context switches. Moreover, XDP facilitates distributing the processing power of the CPU cores in a simpler and more granular way between NFs and applications, particularly when ATR is available.

## 3.7 Discussion and suggested best practices

In previous sections we characterised the performance of the analyzed packet processing technologies on homogeneous classes of traffic (either dropped, redirected out of the machine or delivered to a local application). While for each one of the above categories we were able to identify the most efficient processing technology for the NFs, a one-size-fits-all winner does not exist. Hence, this section leverages previous results to provide some guidance for real world deployments, focusing on resource-limited edge datacenters in which each server may need to handle all the three types of traffic (dropped, forwarded, terminated locally) at the same time, with the highest efficiency. To do so, we derive two preliminary guidelines that could

drive the design and optimal placement of NFs, starting from the simple combination of the takeaways presented in the previous sections, which assumed the presence of a single class of traffic. Then, in the following sections we verify whether these guidelines hold also in real world scenarios, with a combination of traffic of different classes.

- **Tentative guideline 1.** When handling only pass-through and dropped traffic, process it in user space with the system call-triggered driver (*AF_XDP sysc*), thanks to its higher performance (which is even more evident when a huge amount of memory is requested, due to its higher efficiency in that case) and superior processing freedom (no eBPF limitations) (**Takeway 2**). In addition, offload only the most accessed packet dropping rules in the kernel (as long as they fit in the L2/L3 cache), in order to leverage earlier packet discarding without incurring in the memory penalty of XDP (**Takeway 1**), while the rest is left to user-space.
- **Tentative guideline 2.** When handling also local traffic, process this class of traffic in the kernel, to avoid the expensive crossing of the user-kernel barrier multiple times (**Takeway 3**). However, in case a NF needs to operate on all types of traffic, this may require to duplicate its logic both in user space and XDP, which might not always be possible due to the limitations of eBPF. In this case, the developer could evaluate whether (other) existing kernel networking facilities (such as `qdisc`, `netfilter`, etc.) can be used to achieve the desired function, overcoming the limitations of eBPF. Alternatively, he can move local traffic to user space, incurring in the additional cost of re-injecting packets into the kernel to send them to the application through the TCP/IP stack, unless he can modify general-purpose applications to receive the TCP/IP traffic directly from user-space.

### 3.7.1 Mixing pass-through and dropped traffic

To evaluate the effectiveness of **Tentative guideline 1** we defined a NF chain composed of our firewall followed by the load balancer. We compared the performance of the chain running (*i*) purely in user-space (i.e., AF_XDP), (*ii*) purely in kernel-space

(i.e., XDP), and (*iii*) in hybrid mode, in which the firewall logic is at the XDP level and the load balancing logic sits in user space[10].

In this first test, which does not include local traffic, we generated a total of 11K flows: 1K matched the ACL of the firewall (hence were dropped), while the remaining 10K reached the load balancer. Figure 3.21 shows the throughput globally handled by the chain (both dropped and redirected packets) varying the share of traffic belonging to each class. Curiously, the performance advantage of user-level processing with respect to XDP when all packets are redirected decreased from 33% when only the load balancer was used (Figure 3.15) to 19% in this case, suggesting that the user space code seems to be more affected by the additional ACL lookup than XDP. Unfortunately moving the firewall logic in the kernel (*Hybrid sysc* in Figure 3.21) highly impacted the performance of the chain also in the scenario where all traffic was redirected (20% throughput reduction with respect to *AF_XDP sysc*), making even pure XDP processing more effective than the hybrid approach. The performance advantage of both XDP and the Hybrid approach over full user space is noticeable only when the share of dropped traffic exceeds 75%. While in this scenario, with huge amount of dropped traffic, XDP performs slightly better than the Hybrid solution, the latter is much more suitable to a be enabled dynamically, hence allowing to switch from pure AF_XDP to Hybrid upon necessity. In fact, this requires only to replace the existing XDP program with a new one (an atomic operation without service disruption), while switching from AF_XDP to XDP requires creating/destroying sockets and/or changing the interrupt configuration of the NIC queues, which can hardly be done with the current technology.

**Takeaway 4:** When handling only pass-through and dropped traffic, process all traffic in user space under normal conditions (i.e., when most traffic is forwarded) and dynamically offload the most accessed packet dropping rules in the kernel (as long as they fit in the L2/L3 cache) when the amount of dropped traffic is predominant, for example during the mitigation of a DDoS attack.

[10]Our proof-of-concept implementation repeats the parsing of the packet both in kernel and user space. A possible improvement would be leveraging XDP metadata to share information already computed in the eBPF program with the user space [57], whose evaluation is left as a future work.

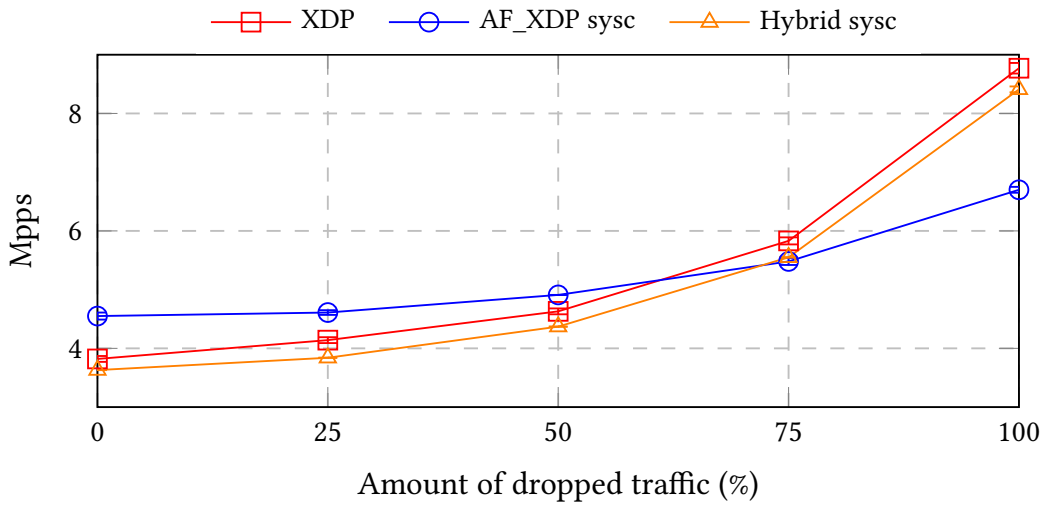


Fig. 3.21 Pass-through + dropped traffic: effect of kernel offloading of packet dropping logic (Hybrid sysc) if compared to pure user space (AF_XDP sysc) and pure in-kernel (XDP) processing.

### 3.7.2 Mixing pass-through and local traffic

The **Tentative guideline 2** speculates that the best choice for pass-through traffic is AF_XDP in user space, while local traffic stays in kernel with XDP. However, to have both the above technologies running at the same time, we need some processing logic that analyzes incoming traffic and redirects each packet to the appropriate pipeline. In this section we assume to have both pass-through and local traffic and we evaluate the feasibility and effectiveness of this hybrid approach, analyzing the different options for the processing logic that separates the two classes of traffic.

First, we analyze a software-based solution running at the XDP level, which redirects part of the traffic in user space for AF_XDP processing, while the rest continues along the TCP/IP stack (`XDP_PASS` return code). Then, we will explore an hardware-based alternative, which leverages the capabilities of the NIC to steer different flows to different receive queues. These tests used a load balancer NF, which redirects local traffic to the proper pod replica of the final application (running on the server itself), while the pass-though traffic was redirected to multiple external servers. In case of traffic splitting, the load balancing logic of each instance will operate only on the portion of traffic handled on that path (e.g., either local or pass-through).

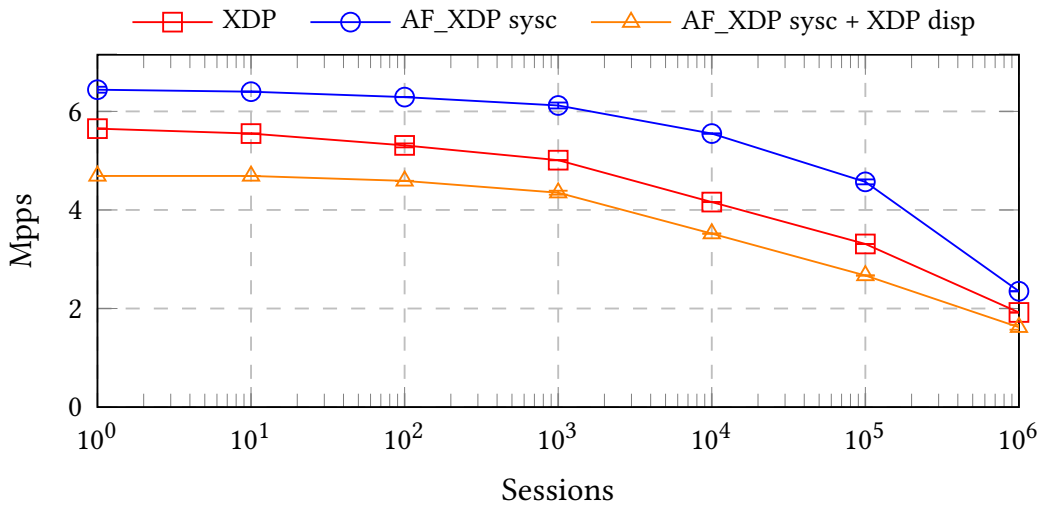


Fig. 3.22 Pass-through traffic only: impact of additional in-kernel packet dispatching logic applied to AF_XDP sysc (AF_XDP sysc + XDP disp) compared to pure XDP or pure AF_XDP sysc.

### Software-based splitting

In this chapter, we assume that the traffic terminating locally can be detected by simply checking the 5-tuple of the packet, which enables the detection of traffic directed to local destinations with a simple lookup on a hash table. To assess the feasibility of this solution we started by measuring the overhead of the additional XDP dispatching logic on the traffic redirecting performance of the *AF_XDP sysc* load balancer, in the optimal case in which the hash table of local sessions contains only one element that is never hit (we generated only pass-through traffic). Results in Figure 3.22 show that the overhead of this solution (*AF_XDP sysc + XDP disp*) greatly affects overall performance, compromising the benefits of user space processing (*AF_XDP sysc*) and making even a pure *XDP* implementation of the load balancer more appealing in case part of the traffic reaches local applications through the TCP/IP stack. This result does not surprise since the dispatching logic has almost the same complexity as the load balancing program, and it is not needed in case of pure XDP. Since the addition of the required splitting logic reverted previous results and made XDP a better solution even in the pass-through-only scenario, which is the preferred battlefield for AF_XDP, we avoided additional tests with local/dropped traffic that, as confirmed by our previous tests, are already pushing further for an XDP-based solution. Nonetheless, a pure software-based splitting mechanism may

still be useful, for example when eBPF limitations prevent the implementation of the correct logic at the XDP level, and therefore we need to rely on slower kernel network stack facilities (e.g. the TC layer, Netfilter). In this case the performance benefit of AF_XDP sockets over the kernel stack for pass-through traffic may overweight the cost of flows dispatching.

**Takeaway 5:** When handling also local traffic (in addition to pass-through/dropped one), and no hardware-based packet steering mechanism is available, process all the traffic in the kernel with XDP, given the prohibitive overhead of software-based packet dispatching.

### Hardware-based splitting

An alternative to software (XDP-based) flows dispatching is to rely on the capabilities of the NIC to steer different flows to different receive queues, leveraging the Ethernet Flow Director technology (Section 3.2.3). This greatly simplifies the operations of the XDP splitter, which can now operate on the (simpler) input queue instead of the 5-tuple. To evaluate this option we extended our setup with an additional traffic generation machine (with the same configuration described in Section 3.3) and connected all of them with a 40 Gbps switch. The first traffic generator leveraged MoonGen to create pass-through traffic, while the second traffic generator sent requests to `memcached` running on the DUT (local traffic). As in the former test, we leveraged the load balancer as a sample NF and selected a pool of 4 CPU cores, allocated in different ways depending on the test configuration. Since pure XDP proved to be the most effective solution for local traffic (Section 3.6), we leveraged the XDP implementation of the load balancer as a baseline, relying on RSS and ATR to 'naturally' distribute all traffic (both local and pass-through flows) across all four available cores, that were therefore shared between NF and application (`memcached`) processing. As an alternative we experimented with an hybrid solution, partitioning our cores in a first set dedicated to pass-through traffic and running the load balancer in AF_XDP mode (with system-call-triggered driver) and another set executing `memcached` and processing packets in-kernel at the XDP level. We used Flow Director rules to instruct the NIC to steer all pass-through traffic to the first set of queues/cores, leaving RSS/ATR to balance local flows on the second set. Since the driver of our NIC supports a single XDP program running on the interface, we added

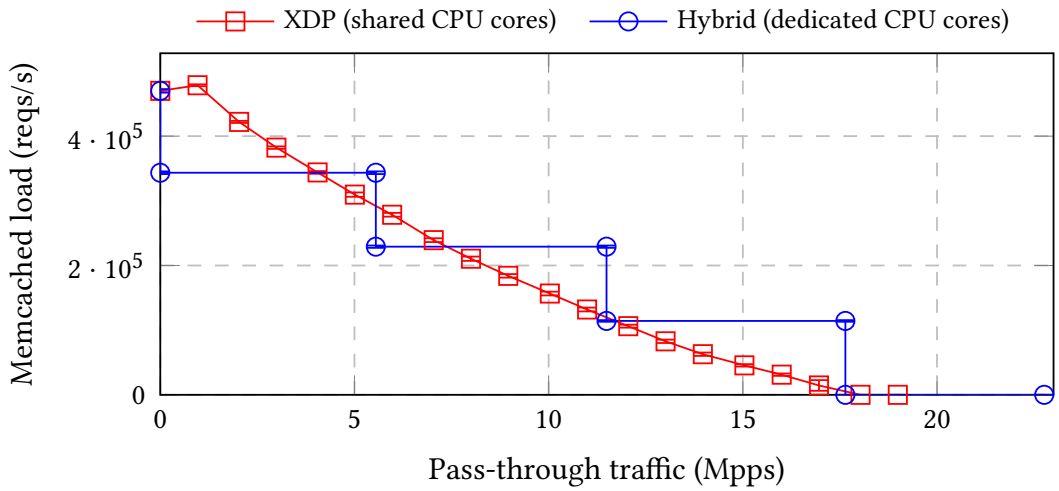


Fig. 3.23 Pass-through + local traffic: effect of hardware-based splitting of local and pass-through traffic processing between kernel and user space (Hybrid) compared to pure kernel processing (XDP).

a simple dispatching logic to the XDP load balancer, using the input queue to decide whether to redirect traffic to user space or to proceed with in-kernel processing.

Figure 3.23 shows the maximum number of requests per second handled by `memcached` (local traffic) for an increasing amount of pass-through traffic hitting the server. The *Hybrid* configuration presents a step behavior, with the performance of `memcached` remaining constant regardless of the amount of pass-through traffic (due to the dedicated CPU cores), until we need to reallocate a core from the set dedicated to local traffic to the one dedicated to pass-through traffic to accommodate the increasing offered load. On the other hand, *XDP* processing proves again to be the most flexible solution with respect to resource allocation, with processing power gradually being re-allocated from local traffic / application to pass-through traffic, although the pass-through traffic seems to have the precedence over local traffic (i.e., the system tends to process all incoming pass-though traffic, at the expense of an inferior number of `memcached` served requested). Despite this increased flexibility, XDP outperforms the *Hybrid* configuration in some operating conditions characterised by a low amount of pass-through traffic (particularly, when the XDP line sits above the Hybrid line in Figure 3.23). When the pass-through traffic exceeds 7.5 Mpps, the *Hybrid* solution provides consistently higher performance to *memcached* with an equal amount of pass-through traffic hitting the server, and

allows to achieve a higher global throughput when all cores are dedicated to pass-through processing.

It is worth mentioning that we faced some limitations in implementing the hybrid solution due to the hardware at our disposal. Flow Director rules available on our NIC only allow to perform exact match on packets (no wildcards allowed) and to steer traffic to a single queue. Therefore, to guarantee a fair load distribution among pass-through cores we had to generate a specific pattern of pass-through traffic. Some modern NICs provide wildcard rules able to steer packets to an *RSS context* operating on a set of queues, so that further load balancing can be applied. A second problem was related to ATR, that is automatically disabled as soon as Flow Director rules are manually added to the NIC. Since ATR internally relies on Flow Director rules automatically generated by the NIC driver, this prevents clashes with user-provided rules. A possible solution would be re-implementing the ATR logic by sampling egress packets, for example with a TC eBPF program. Nonetheless, despite the absence of this acceleration, the Hybrid solution was still able to outperform pure XDP by a good margin.

**Takeaway 6:** When processing all kinds of traffic (i.e., pass-through, dropped and local) resort to in-kernel processing with XDP if the fraction of pass-through load is low. If (*i*) hardware dispatching mechanisms are available, (*ii*) a more complex and rigid partitioning of resources (e.g., CPU cores) is acceptable and (*iii*) a high volume of pass-through traffic is handled, split pass-through and local traffic between user space and in-kernel processing to achieve best overall performance.

For the sake of precision, the above experiment assumes that the traffic splitting operation is simple, e.g., based on IP destination addresses or the session 5-tuple. In some cases, e.g., edge clusters running 5G mobile core components, the traffic processing may include some heavy operations (e.g., User-Plane Function - UPF, GTP de-tunneling) in order to derive the above parameters, raising the question where to implement the above (expensive) processing. The analysis of the above case is left for future work.

## 3.8 Related work

Different works in the recent literature studied the properties and characterized the performance of in-kernel and user space packet processing with a focus on the XDP and AF_XDP technologies.

In [58] Hohlfeld et al. analyze the performance of offloading packet processing both to the kernel and to a SmartNIC leveraging XDP. While advantages and drawbacks of these two approaches are thoroughly studied, the paper does not analyze the possibilities of interaction between in-kernel and user space processing. In [37] authors propose an architecture for eBPF based NFs called eVNF. This architecture encompasses a fast path executed at the XDP level to carry out simple but critical tasks, while a slow path, based either on the kernel network stack or on a user space application accessible through AF_XDP sockets, handles corner cases. Both these papers however, probably due to the early stage of development of the AF_XDP technology, do not consider user space packet processing as a valid fast path alternative.

In [59] authors present the maintainability, flexibility and performance considerations that led to the selection of AF_XDP as the base for the future data plane of Open vSwitch. The paper identifies the performance limitations of user space when processing traffic directed to containers leveraging the TCP/IP stack, and provides some preliminary considerations on the possibility to rely on in-kernel processing for this type of traffic. Our work extends these considerations. [60] proposes a different solution to the problem, experimenting with a user space implementation of the TCP/IP stack to support standard applications and layer 7 NFs.

[61] makes the case for automatically decomposing eBPF/XDP based NFs in multiple programs and between in-kernel and user space components to bypass the limitations imposed by the eBPF verifier.

### 3.8.1 DPDK user space drivers

The most common solution for high-speed packet processing is currently DPDK coupled with custom user space drivers, which enable direct access to the hardware from user space. While the DPDK can leverage AF_XDP and standard Linux drivers for packet I/O, user space drivers present multiple advantages with respect to

AF_XDP, at the cost of taking complete control over the NIC, which is no longer visible by the kernel. In fact, an all-userspace design enables higher performance because it avoids expensive system calls and/or context switches. Furthermore, unlike AF_XDP programs, user space drivers can access many hardware offloadings features such as TSO and GRO, as well as packet metadata (e.g., checksum) provided by modern NICs[11].

However, we did not consider the DPDK technology in this chapter because it is more appropriate for dedicated servers, which may not be the case of small edge-based data centers as well as cloud-native deployments, as suggested also by [59]. DPDK requires the use of memory hugepages, and the `mmap()`ing of large contiguous memory areas, an operation that could fail. Both these constraints complicate the coexistence with other traditional applications. Moreover, user space drivers would prevent native traffic processing in the Linux TCP/IP stack, given the full control of DPDK over the NIC, requiring to re-inject packets from user space into the kernel, whose (low) performance is expected to be similar to the one showed in Figure 3.17. On the other hand, AF_XDP can easily enable part of the traffic to be natively processed by the Linux kernel TCP/IP stack, with traffic steering done either in software (with an XDP program) or with rules installed on the NIC.

A possible way to enable a similar behavior with DPDK would be leveraging SR-IOV with user space drivers bound to one or more Virtual Functions (VFs), the kernel bound to the Physical one (PF), plus the proper additional logic at the NIC level to steer packets among the two as in Section 3.7.2. This logic however might not be supported by the NIC, as in the case of our Intel XL710, where each VF must have a different MAC address used for packet steering, that would make VFs and the PF behave like completely different interfaces.

Given the above considerations, it is worth noting that, in a context in which no traffic has to be processed locally and packets are either dropped or forwarded, DPDK's user space drivers can represent a more efficient choice with respect to the results presented in Section 3.4 and Section 3.5, which builds on the assumption of having shared server. We leave a holistic evaluation, encompassing all available technologies, as a future work.

[11] An attempt to support the above features in AF_XDP was proposed in [62].

## 3.9 Conclusions

This chapter presented the performance characterization of in-kernel and user space packet processing based only on the recent kernel-provided XDP/AF_XDP infrastructure, without the need to maintain and integrate custom kernel modules. Our analysis focused on the scenario of telco edge data centers, where the processed traffic includes both a pass-through portion, handled by a chain of NFs and redirected towards a remote destination, as well as a local portion, directed to applications running on the same set of servers and leveraging the local TCP/IP stack.

We carried out a set of experiments studying these classes of traffic, with the aim of optimizing the usage of (scarce) edge resources by running both data plane-oriented workloads and traditional applications on the same (shared) servers. Our results underline which technology is best suited for each possible mix of traffic, deriving six guidelines that can help telecom operators to select the best technology based on the actual operating conditions, and to achieve optimal performance in a mixed workload scenario such as the one present in the data centers at the edge of the network.

# Part III

# Enabling secure and efficient execution of serverless workloads on multi-tenant servers

# Chapter 4

# SURE: Secure Unikernels Make Serverless Computing Rapid and Efficient

## 4.1 Introduction

Online interactive services are evolving to composable, *loosely-coupled* microservices [63, 64], where each microservice is more quickly developed, tested, and deployed independently. The request from an external client triggers a series of calls between dependent microservices, described as a *call graph* [63]. Serverless computing, often referred to as Function-as-a-Service (FaaS [65]), is a natural fit for loosely-coupled microservice because of the fine-grained billing ("pay-as-you-go") and the unlimited elasticity provided by a serverless infrastructure, thus simplifying the management of the application [66, 67, 68, 69, 70].

**Serverless support for loosely-coupled microservices.** With serverless computing, loosely-coupled microservices can be organized into a "function chain", following their call graph dependencies [3, 63, 71, 66]. As depicted in Fig. 4.1, serverless computing has three important infrastructural building blocks to support loosely-coupled microservices: (1) *Inter-function networking* for communication between decoupled functions; (2) A *service mesh* (*e.g.,* Istio [1]) transparently facilitates the orchestration (observability, traffic management, and access control) of serverless functions in distributed environments by attaching an *individual* sidecar to the

serverless function [2]; (3) A *virtualized function runtime* (sandbox) at the individual microservice (function) level is needed for fine-grained isolation in public clouds.

***State of the Landscape.*** Despite serverless computing bringing many positive capabilities, current platforms have high latency (millisecond-scale), and overheads. Kernel-based inter-function networking and a heavyweight service mesh using a loosely-coupled userspace sidecar are significant contributors [3, 72, 2]. While existing works have sought to improve on these aspects of serverless computing [3, 73, 66], they all rely on containers for their agility. However, containers are vulnerable to an ever-increasing number of exploits due to the expanding attack surface of kernel APIs [74]. Production serverless platforms often use more secure sandboxes by reintroducing virtualization based on userspace kernels [75] or virtual machines (VMs) [76, 77], but this can result in lower performance and slow function startup.

Recent efforts [78, 79] towards secure, lightweight virtualized function runtimes for serverless computing seek to reduce a VM's footprint by stripping unnecessary libraries and drivers from the OS. They create a single address space library OS (LibOS [80])-based function runtime - a *unikernel* [81, 82, 83, 84]. Unikernels are at least as lightweight as containers, boot faster than traditional VMs, and also have the strong isolation of traditional full-size VMs [85]. Further, the entire software stack of a unikernel can be specialized, typically resulting in a much smaller Trusted Computing Base (TCB) and potentially fewer vulnerabilities [86]. These desirable characteristics make unikernels well suited for secure, lightweight deployment of serverless functions, compared to widely used containers or heavyweight virtualization [78]. Additionally, compatibility with existing applications is managed by introduction of a compatibility layer in the unikernel [87]. Popular container orchestration platforms, such as Kubernetes [88], have also expanded support to interface with unikernels [89]. These efforts make unikernels a production-ready solution, such as with NanoVMs [90].

But, considering the "killer microseconds" [91] that serverless computing needs to address, a naive unikernel-based solution is not yet ready to support loosely coupled microservices. Nor can we stand on the shoulders of existing work due to the deficiencies in their designs themselves, or their incompatibility with unikernels: ***(1)*** Unikernel-based serverless environments [78] face the same problem of slow inter-function networking as with containerized environments. While shared memory processing helps with optimizing containerized environments [3, 73], these designs

only considered a single-node data plane optimization and don't fully address the isolation across functions within the same chain. Cross-node communication still uses kernel-based networking. ***(2)*** The performance and resource consumption drawbacks of using an individual sidecar for service mesh functionality persist. Even eBPF-based acceleration [3] faces limitations in achieving full (L7) payload visibility and corresponding functionality in unikernel-based environments.

This chapter describes a unikernel-based serverless computing framework that strives to operate in the best possible region of the design space to support loosely-coupled microservices, while maintaining isolation. We call our work **SURE**, for **S**ecure **U**nikernels that are **R**apid and **E**fficient. **SURE** deploys the serverless function as a VM-based unikernel (called a **SURE** VM), which offers substantial agility and inherently adapts the VM-based isolation at the granularity of independent functions. **SURE** facilitates low-latency and high-performance inter-function networking through zero-copy communication rather than kernel-based networking. First, leveraging the reliable nature of communication within a single host, it utilizes shared memory processing for zero-copy intra-node direct message exchange, ditching the TCP/IP stack as demonstrated in [3] and [73]. Going beyond those designs, we introduce a zero-copy protocol stack (called Z-stack) to facilitate communication *across nodes*. Z-stack has a full-fledged FreeBSD TCP/IP stack implementation running in the userspace with DPDK to mitigate kernel-related overhead. Importantly, Z-stack seamlessly interfaces with the local shared memory data plane, thereby augmenting zero-copy communication to span multiple nodes. To amortize the costs of busy-polling-based packet processing we have a per-node **SURE** gateway that consolidates cross-node communication (i.e., TCP/IP protocol processing) for all co-located functions in a node.

**SURE** fully takes advantage of the LibOS-based design of unikernels by deploying the sidecar as a library linked into the function code within the unikernel. The unikernel's single-address-space design eliminates boundary crossings between kernel and userspace. It simplifies data exchange between the library-based sidecar and user code by using *internal* function calls. This overcomes the shortcomings of an individual userspace sidecar. The execution of the library-based sidecar is event-driven, just like an eBPF-based sidecar. But unlike eBPF, this design allows us to implement complex L7 sidecar functionality with full payload visibility to enable a full-functional service mesh.

**SURE** pays particular attention to the security vulnerabilities from the use of shared memory processing and running the sidecar as a library residing in the same address space as the application code (details in §4.3.3). Going beyond existing solutions (*e.g.,* [3]) that use group-based security domains to enable coarse-grained isolation at the level of shared memory pools, **SURE** offers more fine-grained isolation at the level of memory pages. This allows us to manage the ownership of shared memory pages for each individual function, which can prevent a misbehaving function from inadvertently manipulating shared memory pages owned by other functions, even if they are in the same security domain. Additionally, we isolate **SURE**'s LibOS modules which are part of the serverless infrastructure and contain sensitive data, such as sidecar statistics, from user code that is typically untrusted in a cloud environment.

To achieve these design goals without introducing expensive overheads such as system calls, **SURE** chooses to use Intel's Memory Protection Keys (MPK) [92] for its lightweight memory isolation. **SURE** protects the memory pages storing data of trusted components (i.e., the shared memory data plane and the library-based sidecar) with a privileged MPK key which prevents access from untrusted user code. We provide a "call gate" abstraction for the user code to invoke the *secure API* provided by our trusted LibOS modules. The call gate uses MPK to grant access to protected pages, guaranteeing that only trusted code can operate on them. Moreover, user code is selectively enabled to access shared memory pages containing messages owned by the current VM, allowing overhead-free direct access. Despite that, a single-address-space unikernel where all code runs at *supervisor* privilege level means the untrusted user code is capable of causing potential MPK privilege escalation and gaining unallowed access to protected pages. The root cause is access to core kernel components such as the scheduler, the paging API, and the interrupt service routine infrastructure, which on a traditional OS would be protected by the user/supervisor privilege separation. These components involve sensitive data structures and allowing arbitrary access to their internals can compromise MPK-based isolation. **SURE** moves these components in its TCB and protects them to prevent such exploits (see §4.5.2).

***Summary of Contributions.***

- **SURE** uses event-driven, shared memory processing as in [3], while retaining the philosophy of serverless computing. But **SURE** makes key extensions

to shared memory processing, including connection management and back-pressure for lossless communication between functions.

- **SURE** enhances existing designs [3, 73] by extending zero-copy communication to go across nodes and integrates zero-copy TCP/IP processing with the local shared memory data plane for intra-node communication.
- Our use of a library-based sidecar fully exploits the single-address-space benefits of a unikernel, resulting in more than ***100***× CPU cycle savings and 16× performance improvement compared to the individual userspace sidecar.
- **SURE**'s MPK-based call gates allow running trusted components within the unikernel, benefiting from low overhead interaction with user code while retaining strong isolation. By managing the ownership of each shared memory page, **SURE** exploits the high performance of shared memory processing while avoiding the vulnerabilities introduced by this processing model.

## 4.2 Background and motivation

We start by examining serverless architectures, understanding the challenges that remain necessitating **SURE**. We also discuss the work related to **SURE**.

### 4.2.1 Isolating Serverless Functions

Serverless computing improves overall efficiency and reduces operational costs by enabling different users to share the same cloud infrastructure [70]. This means that different users' functions and network traffic will coexist in the same environment, hence, without proper isolation, there is a risk of unwanted access to sensitive data and resource contention. The use of conventional hardware-level virtualization software (e.g., Xen [93], QEMU [94]) provides strong isolation between co-located VMs as well as flexible resource management by resizing or migrating VMs. However, bare-metal virtualization is heavy since each VM runs an entire operating system and applications. *Containers*, on the other hand, implement OS-level virtualization, which utilizes the capabilities of the host kernel (such as networking stack) and thus does not include the OS kernel of its own. This makes containers more lightweight compared to VMs. However, they are less secure due to their sharing of the host

OS [85, 95, 96, 97, 74]. This has incentivized commercial serverless providers to revert to virtualization similar to VMs to enhance their platform's security [77, 75].

*A Primer on Unikernels.* Using the concept of a LibOS [80], we can rebuild a traditional OS into libraries and bind the application with only the required OS libraries, creating a specialized VM image as a *single-address-space* virtual machine [81] (also called a unikernel). This customization makes unikernels lightweight, with fast startup, while offering stronger isolation than containers [82]. This design also eliminates the kernel-userspace boundary crossing within the unikernel, further reducing the runtime overhead compared to a conventional VM [82].

*Design Implication#1*: Serverless computing requires strong isolation to allow safely multiplexing workloads of multiple tenants on few physical nodes. However, to achieve high density, this isolation cannot come at the cost of agility and leanness. Unikernels strike a good balance between isolation and agility, representing a promising runtime environment for serverless functions.

### 4.2.2 Inter-function networking and service mesh in serverless computing

Fig. 4.1 shows an abstract view of a typical serverless data plane. Essential building blocks include a virtual switch (vSwitch) for L2 forwarding, the network protocol stack (e.g., TCP/IP) for handling application layer messages, and virtual device interfaces (vDevices) to interface the virtualized functions and the vSwitch. A separately running sidecar is attached to each serverless function, connected via the local loopback interface [2]. Depending on the virtualization option chosen (e.g., containers, VMs), the exact building blocks in Fig. 4.1 differ. Common vDevices include veth pairs (for containerized environments) and virtio/vhost devices (for VM-based environments) [98].

#### Cost of kernel-based inter-function networking.

Most of the data plane overheads for function chains: *e.g.,* data copies, context switches, interrupts, protocol processing and serialization/deserialization, come from kernel-based networking [3, 99]. There is duplicate processing at different layers, even if functions are co-located on the same node [3].

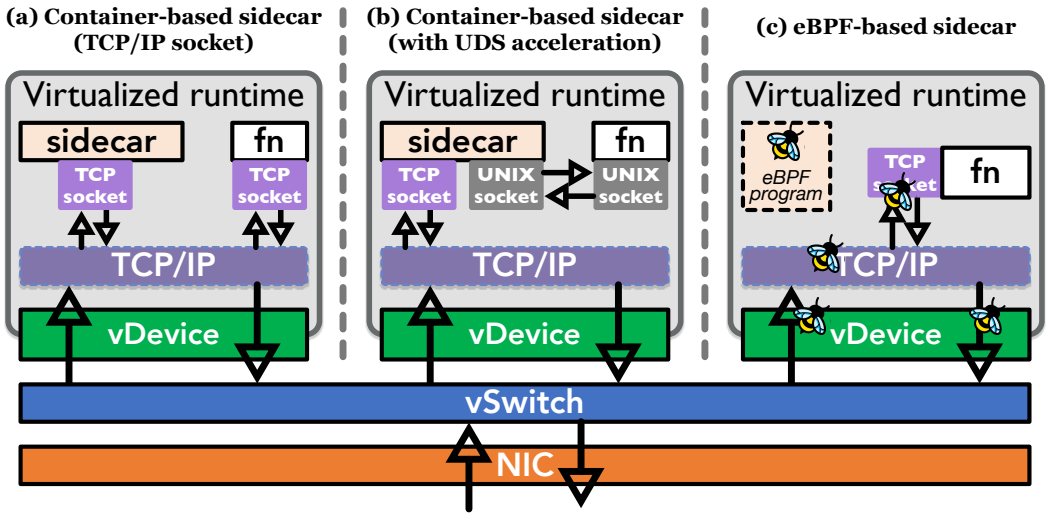


Fig. 4.1 An abstract diagram of serverless support for loosely-coupled microservices. We list existing sidecar designs: (a) container-based sidecar using TCP/IP socket [1, 2], (b) container-based sidecar using UDS acceleration [2], (c) eBPF-based sidecar [3, 4].

**Existing solution: shared memory processing.** [3, 73] design a high-performance data plane for serverless function chaining using shared memory, enabling *zero-copy* communication between functions without incurring any kernel networking overheads.

***Zero-copy networking is limited to intra-node communication.*** The *zero-copy* shared memory communication in [3, 73], is limited to a single node. For cross-node communication or when functions need to interact with external clients/servers, [3, 73] still depend on kernel-based networking. Both [71] and [100] suggest maximizing locality in workload placement to reduce cross-node communication which is not always feasible. Production workloads may contain complex and large-scale microservice call graphs. E.g., Alibaba reports a trace analysis showing that more than 3000 microservices in their workload have interdependencies [63]. Functions can also be resource-intensive (as in data analytic jobs [71]) and need to be spread across multiple nodes.

*Design Implication#2*: *Zero-copy* networking should be extended to inter-node communication. It is important to reduce kernel-related overheads to achieve high performance for cross-node communication.

**Shared memory processing is considered not safe.** Shared memory processing can become a potential conduit for data leakage and corruption [101, 102, 103]. [73]

assumes functions from different users are never co-located, thereby, functions on the same node are from the same user and can trust each other. [3] considers group-based separation, assuming the same user's functions trust each other. Different users have separate memory pools, even in the same node. However, even if functions operating on the same shared memory pool trust each other, an attacker gaining control of one of the functions (by exploiting a vulnerability) might still leverage the shared memory data plane to access and manipulate data belonging to other functions.

*Design Implication#3*: A naïve shared memory implementation that grants access to the whole shared memory pool can compromise functions isolation. Carefully granting access privileges at the individual message buffer level is needed to provide the same isolation level as kernel-based networking. However, this operation must be cheap enough to retain the performance advantages of shared memory processing.

**Cost of individual userspace sidecars.**

Service meshes contribute to the success of microservices [3, 100, 104, 72], the most popular function deployment paradigm in serverless computing. A service mesh acts as a configurable infrastructure layer that facilitates interaction between different functions. In the service mesh, each function is accompanied by a sidecar proxy that deals with aspects related to monitoring, routing, load balancing, and access control [105, 106, 107, 108]. By leveraging the capabilities of the sidecar proxy, a service mesh can effectively manage user functions without being tightly coupled to them. Existing service mesh designs deploy a sidecar as an individual component (*e.g.,* container), independent of the user function. Communication between the individual sidecar and the user function needs to traverse the TCP/IP stack [2] (Fig. 4.1 (a)), incurring unnecessary networking overheads in the data plane [2, 3, 72]. Acceleration includes using Unix domain sockets [109] to redirect the payload between the user function sockets and the individual sidecar (Fig. 4.1 (b)), bypassing protocol processing. But, running an individual sidecar still incurs data copy and serialization/deserialization overheads.

**eBPF streamlines the Service Mesh.** eBPF-based acceleration [4] has been used in containerized environments to provide the service mesh functionality [3, 110] replacing individual sidecars (Fig. 4.1 (c)). eBPF-based sidecars are developed

as run-to-completion programs attached to in-kernel eBPF hooks (e.g., XDP [20], TC [111], and SOCK_MSG [3]). Executed in the kernel, the eBPF-based sidecar avoids the frequent userspace-kernel boundary crossing and duplicate overhead of inter-container communication with the individual userspace sidecar. The execution of the eBPF-based sidecar is triggered upon events, which makes it particularly suitable for event-driven serverless computing [3].

***Limitations of eBPF-based service mesh.*** Despite its flexibility, eBPF does not allow completely custom processing, due to the need to verify programs injected into the kernel to prevent compromising its functionality. The size of programs is limited and operations such as unbounded loops or dynamic memory allocation are forbidden. This can make tasks such as parsing arbitrary (L7) messages very complex, if not impossible [112]. Moreover, the use of eBPF in a unikernelized environment is not a straightforward operation. Hook points providing direct access to messages exchanged on sockets (e.g., SOCK_MSG) are not available since sockets reside inside the guest OS, while intercepting traffic at the packet level can be challenging due to the need to handle aspects such as packet reordering or retransmission.

*Design Implication#4*: An ideal sidecar design should combine the message processing flexibility of a user space sidecar with the low overhead of an eBPF-based one. These properties cannot come at the expense of the isolation of the sidecar from user code, since serverless providers rely on the integrity of this component for service operation.

### 4.2.3 Related work

We have discussed prior work [3, 73] for their limitations in secure shared memory processing and high-performance inter-node communication. The eBPF-based sidecar in [3] is also not viable for an unikernel-based serverless environment. Both [113] and [78] use unikernels as the runtime for serverless functions. But, they lack support for zero-copy communication and a lightweight service mesh, making them less suitable for supporting decoupled microservices. No intra-unikernel isolation in [113] and [78] also is a concern.

**Unikernel/LibOS Virtualization:** Jonas et al. [69] observed a growing demand for fine-grained isolation in serverless computing and identified unikernels as a potential solution to minimize the attack surface. Several past works have optimized

different aspects of unikernels, including system development kits [84, 82, 80, 114], multi-process support [115, 116, 117], fast startup [118, 85, 119], TCP proxies for connection acceleration [120]. **SURE** can take advantage of these unikernels' startup optimizations [118, 85, 119] to reduce the cold-start penalty of serverless computing. CubicleOS [121] and FlexOS [122] offer intra-unikernel isolation using MPK. However, they are not focused on data plane optimization and lack service mesh support, crucial for serverless computing.

**High-performance data plane:** Many high-performance data plane designs have been proposed [9, 123, 124, 125, 126, 3, 118, 127, 128], focusing on performance in a disaggregated environment. Some are designed for NFV, which is less suitable for supporting serverless computing. They lack necessary isolation or memory protection on the data plane [3, 126], or do not support lightweight manageability [124, 118, 127, 128], making them not ideal for serverless computing.

**MPK-based isolation:** Multiple proposals study the application and optimization of Intel's MPK [117, 129, 130, 131, 132, 133]. Jenny [132] filters MPK-related syscalls to prevent unauthorized changes to the MPK key. ERIM [129] enforces binary inspection and rewriting to prevent misuse of MPK. libmpk [133] overcomes the limit of 16 MPK keys by carefully recycling and redistributing keys. These works (ERIM [129] and libmpk [133]) are complementary to **SURE**.

## 4.3 Overview of SURE

### 4.3.1 System architecture of SURE

The goal of **SURE** is to create an appropriately isolated serverless environment while providing high-performance inter-function networking and lightweight service mesh for loosely-coupled microservices. Fig. 4.2 shows the overall architecture of **SURE**, including the following core building blocks: **(1) Unikernel-based function runtime. SURE** employs a unikernel-based function runtime (the **SURE** VM in Fig. 4.2) that strikes an ideal balance between providing isolation and being lightweight. The hypervisor on each worker node is in charge of controlling the life cycle of **SURE** VMs: creating and destroying VMs, allocating addresses, and initializing shared memory. **(2) Intra-node shared memory data plane.** When functions are co-located on the same node, we leverage shared memory processing for

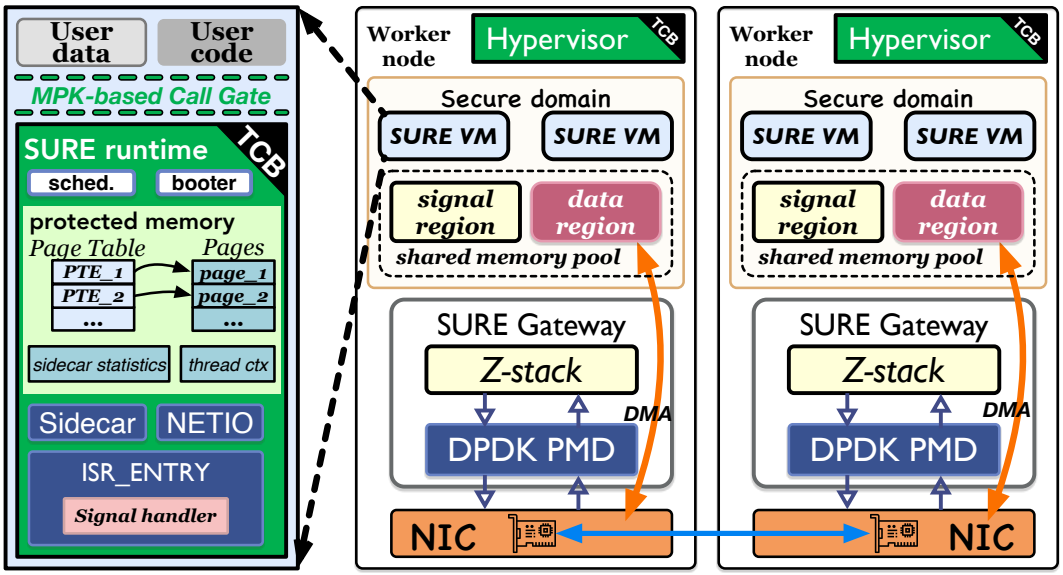


Fig. 4.2 The overall architecture of **SURE**. Note that we only show a single security domain here.

zero-copy intra-node communication (§4.4.1). **(3) A zero-copy protocol stack for the inter-node data plane.** For cross-node traffic, we develop a high-performance, *zero-copy* user-space TCP/IP stack, Z-stack (§4.4.2) — a zero-copy enhancement to the existing DPDK F-stack [18]. Z-stack seamlessly interfaces with **SURE**'s *zero-copy* intra-node data plane. **(4) Consolidated protocol processing.** To amortize the costs of busy-polling-based packet processing, we consolidate TCP/IP processing in a single per-node **SURE** gateway (§4.4.2). **(5) Lightweight library-based sidecar.** As shown in Fig. 4.2, **SURE** moves each sidecar to be a library within the **SURE** VM (§4.4.3). The message exchanges between the sidecar and the functions are simple internal function calls, entirely eliminating a number of data plane overheads of the individual sidecar deployment model.

### 4.3.2 SURE's trust model

**SURE** assumes a one-way trust model typically considered in a public serverless cloud: User functions trust the serverless infrastructure (*i.e.,* **SURE** VM) provided by **SURE**. However, **SURE** does not trust tenants, since applications may contain security vulnerabilities, *e.g.,* buggy code. We assume that functions from the same tenant (*i.e.,* same serverless user) trust each other. But, functions running in **SURE**

are exposed to threats from other, potentially adversarial, tenants in the same cloud. As shown in Fig. 4.2, **SURE** treats the hypervisor and associated toolchains (*e.g.,* emulation of hardware devices and peripherals that are required by VMs) as part of the TCB. We further establish another layer of trust within the **SURE** VM, which is responsible for enforcing intra-unikernel isolation between the untrusted user code and the unikernel TCB modules (scheduler, sidecar, network I/O lib, etc).

### 4.3.3 SURE's threat model

Based on **SURE**'s trust model and system architecture, we identify the following threat sources in **SURE**: **(1) Vulnerabilities from shared memory processing.** Without rigorously enforced access controls, a malicious function might exploit shared memory to gain unauthorized access to sensitive data or perform memory-based attacks, such as Flush+Reload attack [101], buffer overflows [134] or injection attacks [135]. This risk is particularly pronounced in a public cloud environment, shared by functions from different users. In addition, buggy (even if not malicious) code in user functions may accidentally and improperly manipulate shared data. **(2) Intra-unikernel vulnerabilities. SURE**'s function runtime (**SURE** VM), including the library-based sidecar, the code interacting with the shared memory data plane, and other components needed to guarantee security (see Fig. 4.2), are part of the serverless infrastructure and require additional isolation from untrusted user functions. This cannot be guaranteed by the single address space design of a unikernel. A typical threat involves tampering with application-level observability: *e.g.,* buggy function code could inject false metrics into the sidecar, disrupting the service mesh control plane that relies on the integrity of metrics to orchestrate functions.

### 4.3.4 Isolation in SURE

By operating each function within an unikernel-based **SURE** VM, **SURE** leverages the sandboxed environment provided by VMs and the inherent hardware-level virtualization to ensure stronger isolation than containers. **SURE** additionally introduces the following features to enable inter-function access control, protect shared memory processing, and enable intra-unikernel isolation to separate user code with LibOS modules in the unikernel:

**Group-based security domains with isolated memory pools.** We require a "security domain" to be specified when deploying a function on **SURE**. We assign mutually trusted **SURE** VMs (typically from the same user) to the same security domain. Each security domain possesses a private memory pool, established as a POSIX shared memory backend in the host file system, corresponding to a *file*. **SURE** only allows shared memory processing when functions are within the same domain.

**Access control with sidecar and SURE gateway.** Ownership transfer of a shared memory buffer occurs through a descriptor exchange within the security domain. We verify the eligibility of the receiver of the descriptor to prevent unauthorized shared memory access. **SURE** takes advantage of the library-based sidecar to apply traffic filtering (with a whitelist of allowed peers) on the RX and TX paths of the **SURE** VM (Fig. 4.7). The sidecar discards the descriptor if the whitelist does not match. Further, we use the **SURE** gateway to perform a copy between memory pools (on the same node) in different security domains (Fig. 4.3). We also use the **SURE** gateway to enforce cross-node access control by having traffic filtering as part of Z-stack's protocol processing.

**Memory isolation and MPK-based call gates.** While bringing performance benefits, running trusted components such as the sidecar within the same address space as the untrusted user code can compromise their integrity, since buggy or malicious code can easily access and corrupt their internal data structures. The same considerations apply to the shared memory data plane, where the code of a single function has access to the buffers of all services operating on the same memory pool, resulting in reduced isolation. This requires an additional layer of memory isolation to prevent unwanted memory access in **SURE**. Rather than adhering to the conventional approach of kernel-userspace separation and relying on heavyweight system calls (*e.g.,* `sendmsg()`, `recvmsg()`) to guarantee the integrity of trusted components, **SURE** opts for a more streamlined and rapid solution by using Intel's MPK [92] to mitigate the risk of unwanted memory accesses. This effectively retains the data plane performance enhancements of **SURE**, while enabling isolation at the granularity of memory pages. We protect the memory pages storing data of trusted components (including the shared memory data plane and the library-based sidecar) with an MPK key which prevents access from untrusted user code. **SURE** enforces the policy that access to this protected memory initiated by the user code can only be performed through *secure APIs* provided by **SURE**. Functions of these APIs are

wrapped into "call gates" which elevate the privilege through MPK to allow access to protected pages and are the only *controlled* locations where MPK manipulation is allowed. Additionally, user code can selectively be granted direct access to message buffers in the shared memory to remove the overhead of accessing messages. To build a secure isolation mechanism upon MPK, **SURE** also needs to secure some core components of the unikernel, such as the scheduler and the paging API, whose operations also manipulate MPK's foundations (see Section 4.5.2)

**Implementation:** We base the development of **SURE** on Unikraft [84], an automated system for building unikernels. This allows us to reuse key OS building blocks from Unikraft, such as the scheduler, memory allocator, and file systems. **SURE** makes several key enhancements, including a redesigned intra-node and inter-node data plane, the library-based sidecar, and security domain extensions. We use QEMU/KVM as the hypervisor, however, **SURE** can also work with others, such as Firecracker [77].

## 4.4 Data plane design in SURE

### 4.4.1 Intra-node shared memory processing

**SURE** is the first to comprehensively utilize shared memory for unikernels in serverless computing. **SURE** uses efficient, event-driven shared memory processing to match the event-driven execution of serverless functions. **SURE** supports reliable (i.e., no loss, in-order) data transfer for shared memory processing without depending on heavyweight protocol processing such as TCP. Fig. 4.3 shows the intra-node shared memory data plane of **SURE**. VMs in the same security domain on a node share a dedicated memory pool. Each memory pool is composed of pre-allocated buffers (in the *data region*) to store actual message payloads,[1] and a *signal region* for the purpose of descriptor exchanges. By owning a descriptor, functions can access the payload in shared memory directly, thus avoiding data copying.

[1] **SURE** doesn't impose any constraint on the format of exchanged data, allowing the exchange of both serialized content (e.g., an HTTP payload) for decoupled applications, or raw binary data for binary-compatible applications, avoiding the serialization/de-serialization overheads. Applications in **SURE** can also allocate a large enough buffer to store the complete payload, avoiding assembly and disassembly during shared memory data transfer.

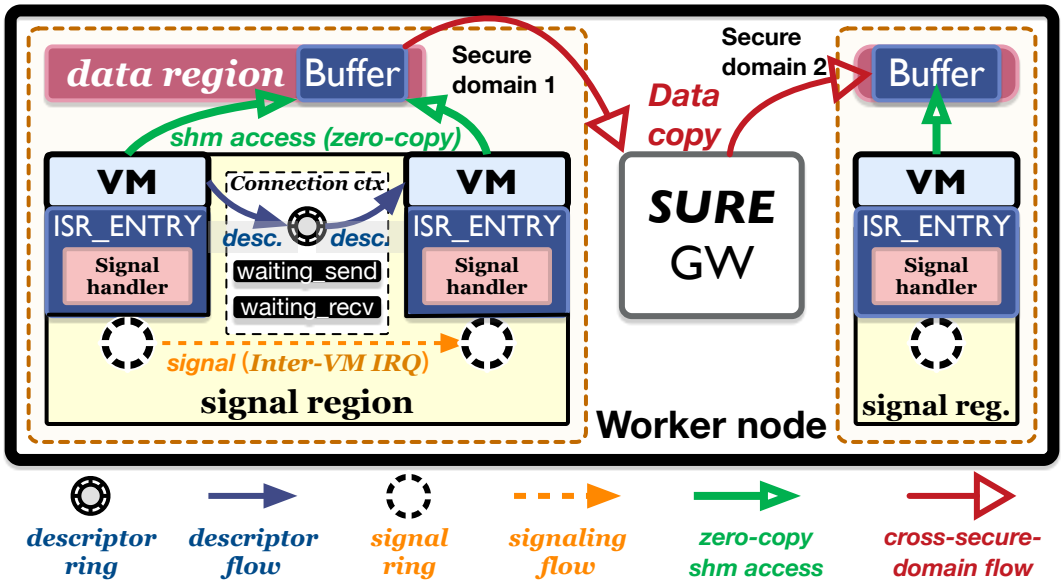


Fig. 4.3 Intra-node data plane in **SURE**. Communication across security domains **within the same node** uses **SURE** gateway (GW) to copy data between memory pools. ISR: Interrupt Service Routine.

### Reliable, in-order message exchange through connection management

**SURE** provides the *connection* abstraction as a bi-directional communication channel between two endpoints, which guarantees *reliable*, *in-order* delivery of messages. As Fig. 4.3 shows, each connection is represented by a "connection context" in the *signal region*, with a single-producer, single-consumer *descriptor ring* used by the receiver to receive the descriptor. This eliminates potential race conditions in descriptor exchanges between a sender and receiver. Entries in the *descriptor ring* are handled in FIFO (first-in, first-out) order. This guarantees the message produced by the sender is always consumed by the receiver in order when *no* loss happens, ensured by the back-pressure mechanism. Back-pressure in **SURE** blocks the sender if receiver's *descriptor ring* is full, indicating the recipient is not consuming descriptors at a sufficient rate.

### Event-driven signaling mechanism

Different operations in **SURE** require a peer to block until a specific event happens (e.g., a message is available). This allows to avoid expensive busy-polling processing, which can result in a waste of resources not acceptable in the serverless context.

Currently, a thread within a **SURE** VM can block because the connection it is receiving from doesn't contain any message, the ring of the connection it is sending to is full (back-pressure) or it is waiting for an incoming connection to be accepted. To support event-driven signaling, each **SURE** VM has a *signal ring* where peer VMs can push the identifier of the thread that needs to be woken up. The *signal ring* is a multiple-producer, single-consumer ring. Before blocking, a thread stores its identifier in a proper data structure (e.g., in the *connection context*), indicating its desire to be woken up. When a peer VM wants to wake up the thread, it pushes its identifier on the *signal ring* of the target VM and optionally sends an inter-VM interrupt, if the ring was previously empty. In the target VM, a *signal handler* is in charge of handling signals. An interrupt service routine is triggered upon reception of inter-VM interrupts and wakes a thread in charge of draining the *signal ring* and waking up the corresponding threads. The use of a dedicated signal polling thread allows to reduce the number of inter-VM interrupts in scenarios where many signals are exchanged. The following sections describe how the signaling mechanism operates when receiving a message and in implementing the back-pressure mechanism.

### Blocking receive

Fig. 4.4 depicts the **SURE**'s event-driven signaling mechanism and how it facilitates the descriptor exchange without active polling: ① The receiver thread calls the `recv()` to consume the descriptor from the *descriptor ring*, but if *no* descriptor is available, ② the receiver thread registers its pointer in the `waiting_recv` field of the connection (via `waiting()`), and blocks. ③ After the sender thread produces a descriptor on the *descriptor ring* via `send()`, ④ it calls `wake_up()` to ⑤ inspect the `waiting_recv` in connection context to check if the receiver thread is currently blocked. ⑥ If the receiver thread is blocked, the sender writes the receiver thread pointer to the *signal ring* of the receiver VM. Optionally, if the *signal ring* was previously empty, ⑦ the sender issues an inter-VM interrupt to the receiver's VM. ⑧ The Interrupt Service Routine wakes the `signal_poll` thread to ⑨ consume all signals (i.e., thread pointers) currently enqueued in the *signal ring*. ⑩ The *signal handler* wakes up the corresponding receiver thread ⑪ to consume the available descriptor on the *descriptor ring*.

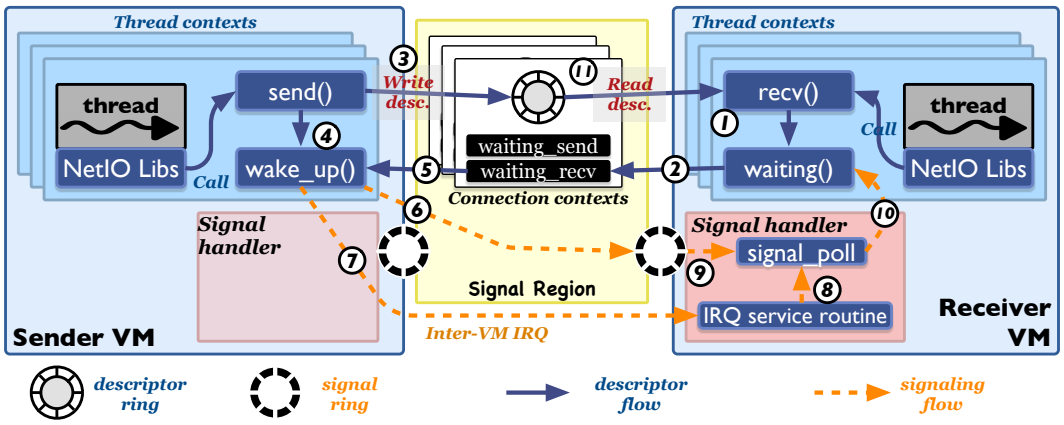


Fig. 4.4 Descriptor exchange and event-driven signaling mechanism between a pair of **SURE** VMs (sender and receiver). The flow depicts the blocking receive in **SURE**.

### Back-pressure

Fig. 4.5 (a) shows the blocking phase of the back-pressure mechanism in **SURE**: ① The sender thread invokes the `send()` API to send the message to the target receiver. However, because the receiver thread is not consuming the descriptors in the *descriptor ring* in a timely manner, the *descriptor ring* can become full. The sender thread will not be able to write the descriptor into the *descriptor ring*. ② The sender thread then invokes the `waiting()` API, registers its pointer in the `waiting_send` field of the connection, and blocks. The back-pressure mechanism ensures the *descriptor ring* never overflows, thus avoiding data loss. Fig. 4.5 (b) shows the processing flow of how the receiver wakes up the sender to suspend the back-pressure mechanism: ① After the receiver thread consumes a descriptor from the *descriptor ring*, it checks the `waiting_send` field for a registered pointer indicating that the sender thread is blocked. ② The receiver thread retrieves the pointer, and ③ writes it to the sender VM's *signal ring*. Optionally, if the *signal ring* was previously empty, ④ the receiver thread sends an inter-VM IRQ to the sender VM. ⑤ The sender VM's Interrupt Service Routine wakes the `signal_poll` thread which in trun ⑥ unblocks the sender thread. ⑦ The sender thread returns to the `send()` and ⑧ writes the descriptor to the *descriptor ring* of the receiver thread.

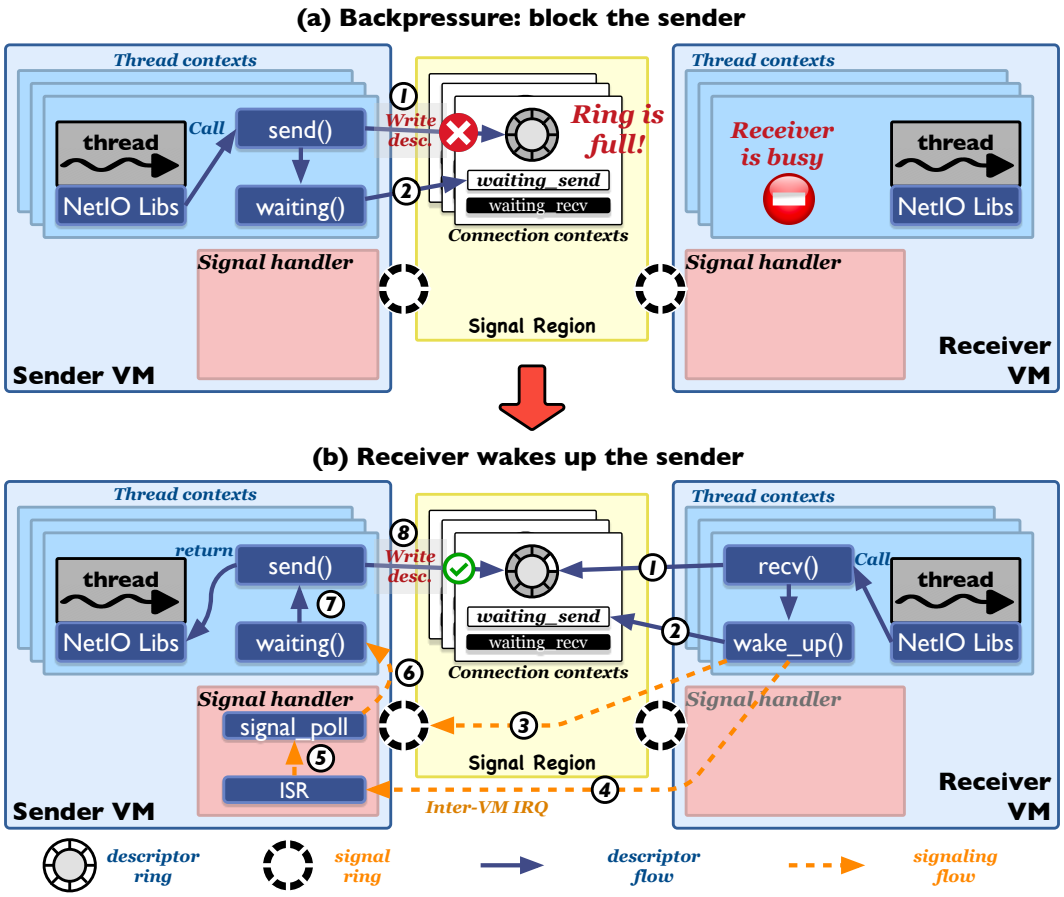


Fig. 4.5 Processing flow of the back-pressure mechanism in **SURE**; (a) the sender block on a full ring, (b) the receiver wakes up the sender.

### 4.4.2 Inter-node communication in SURE

**Consolidated protocol processing.** The single per-node **SURE** gateway provides consolidated protocol processing that is performed once for all incoming/outgoing traffic, with the processed payload residing in the shared memory pool. The **SURE** gateway is integrated with our Z-stack that offers *zero-copy* TCP/IP protocol processing. Combined with the zero-copy intra-node data plane, **SURE** achieves true *zero-copy* communication in and out of the node, unlike previous efforts such as SPRIGHT [3].

**Z-stack: Zero-copy userspace TCP/IP stack.** Like previous efforts, we also seek to bypass the kernel in Z-stack. But, going a further step, we work with DPDK's Poll Mode Driver (PMD [136]) to DMA the packet between the NIC and shared

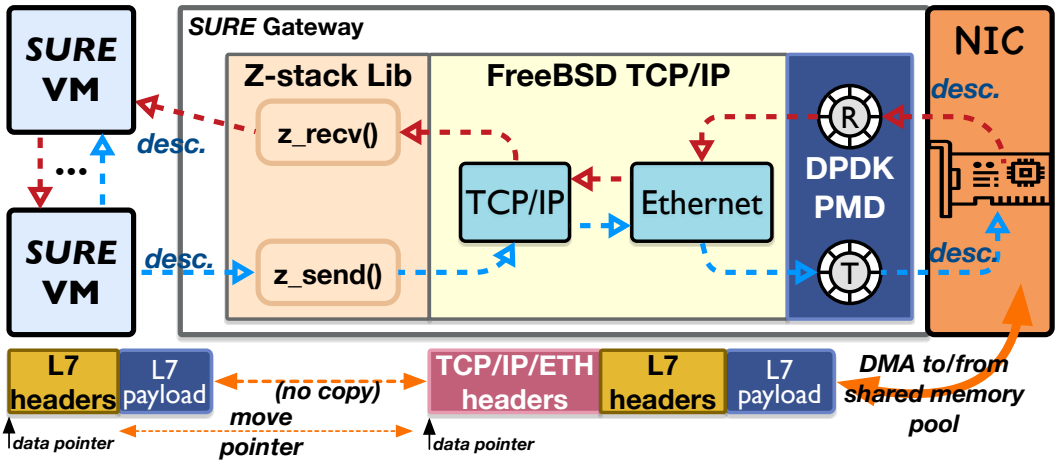


Fig. 4.6 Protocol Processing Pipeline within the Z-stack.

memory in userspace, while also avoiding context switches and interrupts incurred by the kernel protocol stack [99, 137]. In addition, most protocol stack designs rely on the POSIX-style socket interface (e.g., `send()` [138], `recv()` [139]) to interact with user-space applications. This introduces an additional copy when moving the data between the application's send/receive buffer and the socket buffer (accessed by the TCP/IP protocol stack), and can consume more than 50% of the total CPU cycles [99]. Such a design is influenced by the conventional notion of kernel and userspace isolation, i.e., using a copy to ensure data isolation between the user and kernel space. However, it is unnecessary in **SURE** as our data plane operates entirely in userspace.

Fig. 4.6 shows the design of Z-stack. We introduce two new zero-copy APIs (`z_recv()` and `z_send()`) in Z-stack to interface between the application layer and underlying TCP/IP protocol layers by exchanging pointers to the original packet payload. We further add or remove protocol headers by manipulating the data pointer contained in the descriptor, which points to the location where the data begins. Passing the buffer between different layers involves modifications to protocol headers. To avoid recreating buffers, we pre-allocate headroom in buffers that can be filled with headers across the different layers. We base our implementation of Z-stack on top of the existing DPDK F-stack [18], which offers a fully-functional TCP/IP stack ported from FreeBSD integrated with the DPDK PMD. However, F-stack introduces data copies during protocol processing, which we eliminate in Z-stack.

### 4.4.3 Library-based sidecars

We utilize the fact that in LibOS, internal interactions are essentially function calls. This allows us to extend the application in LibOS with a corresponding lightweight sidecar functionality. Instead of having a separate sidecar process running, **SURE**'s LibOS exposes event-based execution hooks for running additional sidecar functions, which has proven successful in eBPF-based service meshes with improved resource efficiency [4, 3]. With this extensible event-driven functionality, **SURE** offers typical sidecar capabilities in its unikernel, including monitoring and traffic management.

**Event-driven execution hooks:** In order to maintain the transparent operation of the sidecar relative to the serverless function, we predefine two hook points each on the Receive (RX) path and Transmit (TX) path, located in `send()` and `recv()` in **SURE**'s network I/O libs (Fig. 4.7) to invoke sidecar functions. **SURE** allows cloud providers to customize the sidecar by adding/removing sidecar functions to/from hooks based on events they are interested in. Required sidecar functions are organized in an execution sequence, driven by I/O events occurring on the RX/TX paths.

**Implementation of library-based sidecar:** We adopt the implementation methodology from the popular service mesh solution, Istio [1], which encapsulates a sidecar functionality into a handler function and organizes the sidecar functions into a call sequence, as shown in Fig. 4.7. We have implemented a set of commonly used sidecar functions in **SURE** (such as request logging, metrics collection, rate controllers, and access control) to enable critical service mesh capabilities, such as monitoring and traffic management.

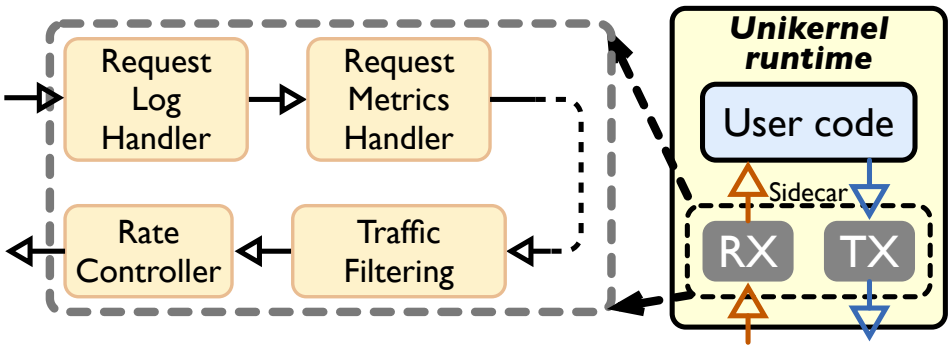


Fig. 4.7 Library-based sidecar in **SURE**. The sidecar contains a sequence of handlers that perform certain sidecar functionalities on both RX and TX data path of the user function.

## 4.5 Memory-level isolation in SURE

In this section, we describe how we use MPK to enable memory-level isolation to protect the shared memory data plane and the single-address-space unikernel.

**A Primer on MPK:** MPK is a hardware-level memory isolation feature introduced in Intel's server CPUs starting from the Skylake microarchitecture. MPK allows tagging a memory page with a key by modifying the corresponding Page Table Entry (PTE). MPK allows for a total of 16 distinct keys to be defined. The access privilege of MPK keys is defined by a per-core CPU register, `PKRU` (Protection Key Register User), where each key is described by 2 bits [133]: (i) "Access Disable" (*AD*) that defines whether access to tagged pages (both read and write) is disabled (bit set to 1); and (ii) "Write Disable" (*WD*), that specifies whether write to tagged pages is disabled (bit set to 1). Pages tagged with the same MPK key can hence be subject to three different access policies: Read/Write (0, 0), Read-Only (0, 1), or No-Access (1, x) [133]. MPK offers a unique x86 instruction, `WRPKRU`, to write the contents of the `PKRU` register, hence switching the access permission for memory pages. While assigning a key to a memory page requires modifying the corresponding Page Table Entry (PTE) and flushing it from the TLB, an expensive operation even without the syscall cost enabled by a unikernel environment, `WRPKRU` allows changing the access privilege to a large set of pages with a single, lightweight instruction.

### 4.5.1 Secure APIs based on SURE call gates

In **SURE**, we use 2 distinct keys out of 16: one associated to unprotected memory (`UK`) and one to protected memory (`PK`). The value of the `PKRU` is always configured to allow access to memory pages tagged with `UK`, while access to pages tagged with `PK` is disabled when executing user code and enabled when executing the privileged code. To execute a privileged function, the user code must go through a ***call gate***, which updates the `PKRU` to enable/disable access to `PK` pages and switches execution of the function onto a stack in protected memory. All other updates to access privilege (*i.e.,* writes to `PKRU`) are illegal, prohibited via the binary inspection of §4.5.2. While most of the memory related to protected components is statically tagged with the `PK`, shared memory buffers are dynamically tagged by changing the key. API functions receiving a message in a buffer or allocating a new buffer update the corresponding key to `UK` to allow unprotected access from user code. When

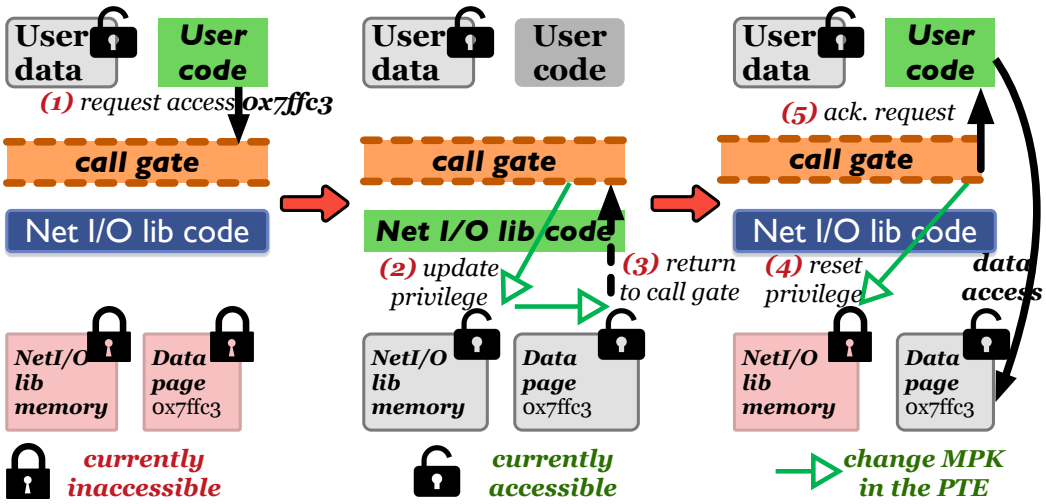


Fig. 4.8 **SURE** uses call gates to secure function calls by user code. **SURE** can dynamically change the access privilege of memory pages.

functions send a message or free a buffer we perform the inverse operation, setting the key to `PK`.

Fig. 4.8 shows how the network API `recv()` function is invoked by untrusted user code via the MPK-based call gate. Other protected API functions are guarded by the MPK-based call gate in the same way. When running user code, the set of protected memory pages (i.e., local data in a **SURE** VM or shared memory) is configured to be inaccessible. ① the user code invokes functions in NetI/O lib via the call gate. ② the call gate adjusts permissions, making protected memory accessible, and invokes the `recv()` function; the NetI/O function receives a buffer descriptor and updates the corresponding MPK key to allow unprivileged user access to the buffer. ③ Upon return to user code, ④ the call gate disables access to protected memory, while the received buffer remains accessible.

### 4.5.2 Preventing privilege escalation of MPK

On its own, MPK is not designed to provide strong security guarantees, being the instruction used to modify the access privilege of keys, `WRPKRU`, an unprivileged user-level instruction. [129] describes how to build a security primitive on top of MPK for a user-space process, by coupling MPK with binary inspection of the code to prevent unwanted modifications of the `PKRU` register. Running its functions in unikernels, **SURE** has to secure three additional systems that in a traditional OS

would be protected by the user/supervisor privilege separation: the paging API; the scheduler; and interrupts. These components operate on the page table or on CPU registers (including `PKRU`), hence they are part of the TCB and their integrity must be guaranteed to prevent unwanted MPK privilege escalation. This is in line with the considerations made by previous works leveraging MPK to provide intra-unikernel isolation [122, 130]. Following are the key components needed to transform MPK into a security primitive in a unikernel environment[2].

**(1) Binary inspection.** To guarantee that only safe occurrences of MPK-related instructions (`WRPKRU` and `XRSTOR`) are present and untrusted code cannot manipulate `PKRU` without passing through our call gate, we couple MPK with binary inspection of the executable, as in [129]. We also enforce a strict write-xor-execute memory policy [142] in **SURE**. This guarantees that executable pages are not modified at run time with instructions changing memory access permissions.

**(2) Paging API protection.** **SURE** protects memory pages by writing an MPK key in their corresponding PTEs. However, PTEs are not protected in the single-address-space unikernel, which means untrusted code may manipulate these PTEs to gain unauthorized access to protected memory pages. This may be done either by changing the MPK key in the PTE or by remapping the (guest) physical address of a protected page to a different, unprotected PTE. To prevent these exploits, **SURE** stores the (guest) physical address of all protected pages in a blacklist. All pages containing a page table are also tagged with the protected key, hence preventing untrusted code from directly modifying the PTEs. To interact with the page table, a trusted API protected with call gates is provided. Whenever a PTE is modified, the API functions check whether the physical address in the PTE being modified belongs to the blacklist (i.e., the physical page is protected) and crash the unikernel in case of a match. This allows untrusted components to modify PTEs describing unprotected memory, guaranteeing that no operations that would compromise protected pages are performed.

**(3) Scheduler protection.** The scheduler is in charge of core operations of the lifecycle of a thread in the unikernel, including context switching that involves storing/loading the thread context into/from memory. The thread context in the **SURE** VM contains the `PKRU` register, directly related to **SURE**'s guarantee of

[2]Note that **SURE** does not offer control flow integrity of MPK, nor protection against side-channel and microarchitectural attacks, which are beyond the scope of our threat model. These issues can be addressed by existing approaches [129, 140, 141].

memory protection. Thus, the thread context must be stored in protected memory throughout its lifecycle (Fig. 4.2). Hence, we run the whole scheduler as privileged code and allow its access from untrusted code only through call gates for safe operation of thread context.

**(4) Interrupt validation.** When an interrupt service routine (ISR) is executed on a CPU, the value of the `PKRU` remains the same as the one associated with the code being interrupted, meaning that interrupting privileged code will cause the ISR to be executed with access to protected memory. To prevent the exploitation of this feature, we extend the entry code common to all ISRs to set the `PKRU` to the unprivileged value upon interrupt entry and restore it to the privileged value upon exit if a privileged function was interrupted. Moreover, when an interrupt is triggered, the CPU stores the instruction pointer (`IP`) and stack pointer (`SP`) of the interrupted task on the *unprotected* interrupt stack, and restores them upon exit to resume its execution. An untrusted interrupt routine could corrupt these values and, in case of interruption of privileged code, cause the return to unprivileged code with access to protected memory. To prevent this exploit, we further extend the common ISR entry code to store the (`IP`, `SP`) pair in a per-CPU protected-memory area when privileged code is interrupted, and to check that the values have not been modified on the stack before returning from the interrupt. As **SURE** only protects the entry/exit points of the ISR, we retain the flexibility for other unikernel libraries (e.g., `virtio-blk` drivers for disk access) to register custom interrupt handlers, without the need to move them inside the TCB and avoiding unintended memory access caused by buggy code in the ISR.

## 4.6 Performance Evaluation of SURE

We quantitatively evaluate the performance improvement and resource efficiency with **SURE**'s data plane. We start with a microbenchmark analysis, to quantify the benefit of each design choice in **SURE**'s data plane. We also evaluate **SURE** with the realistic online boutique workload [143] from Google.

**Testbed Configuration:** We built our testbed on NSF Cloudlab using three nodes, equipped with a 32-core Intel Xeon Silver 4314 CPU running at 2.4 GHz, 128GB of memory, and a 100Gb NIC for network connectivity. The CPU has MPK support. Throughout the experiments, we use Ubuntu 22.04, kernel version 5.15 as the OS.

### 4.6.1 Microbenchmark Analysis

#### Improvement from shared memory processing

We evaluate the round-trip latency and throughput between a client and server pair on the *same* node, to reflect the typical basic interactions between functions. We choose three representative message sizes: 64B, 4KB, and 8KB, since there is very little variation for **SURE** as the packet size goes from 64B to 1KB. We consider the following alternatives that are widely used in commercial and open-source serverless platforms to compare with **SURE**'s intra-node shared memory data plane: (1) Container (denoted `CT`); (2) Unikraft unikernel [84] (denoted `UK`); (3) OSv unikernel [82] (denoted `OSv`). For the alternatives, we use the kernel Linux bridge for L2 connectivity. We additionally consider the userspace Open vSwitch (OVS [35]) for `UK` and `OSv`. To match **SURE**'s reliable data transfer, we use TCP for reliable data transfer with the compared alternatives. Note that container uses the host's TCP/IP stack, Unikraft uses *lwip* [144], and OSv's TCP/IP stack is ported from FreeBSD. For throughput measurements, we add sufficient clients to saturate the server and metrics are collected on the server. To accurately assess the improvements from shared memory processing, we disable **SURE**'s sidecar for this experiment.

***SURE achieves low-latency.*** We measure the latency for a single client-server connection in Fig. 4.9 (left). **SURE** has the lowest latency (14-16us) across all evaluated message sizes. Unlike the other alternatives, **SURE**'s shared memory zero-copy data transfer causes latency to be flat with the increase in message size. We note that *lwip* (used in the Unikraft setup) incurs higher latency as it is under-optimized to work with virtio devices (no checksum offload[3] and extra copy[4]).

***SURE is more scalable and efficient.*** We evaluate throughput (requests per second (RPS)) as the number of concurrent connections increase. Fig. 4.9 (right) shows the RPS for a message size of 64B. Our observations are consistent across other message sizes (4KB, 8KB). Compared to other alternatives, **SURE** is also more efficient: with more than 16 connections, all alternatives have their server's one assigned CPU core saturated. But **SURE** has much higher RPS (and still increases with increasing number of concurrent connections).

[3] https://savannah.nongnu.org/patch/?10111
[4] https://github.com/unikraft/lib-lwip/blob/staging/uknetdev.c#L166-L167

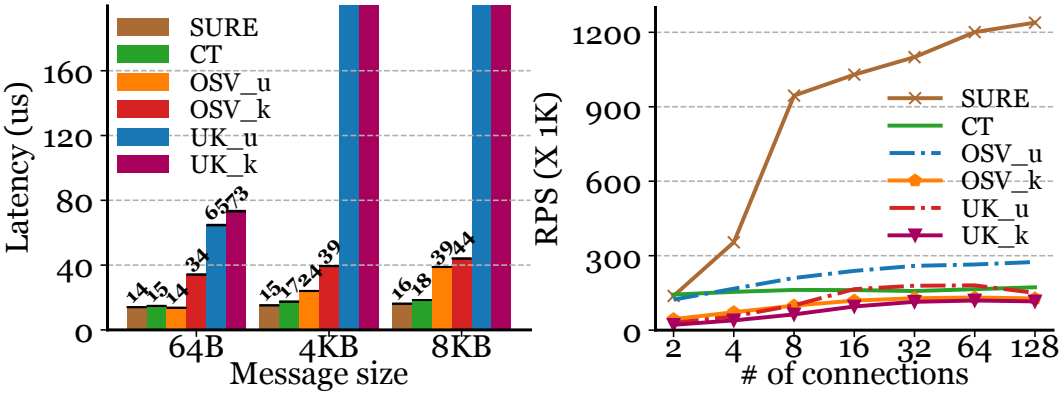


Fig. 4.9 Intra-node data plane performance. `CT`: container with Linux bridge; `OSv_u`: OSv with userspace OVS; `OSv_k`: OSv with Linux bridge; `UK_u`: Unikraft with userspace OVS; `UK_k`: Unikraft with Linux bridge; (average of 30 repetitions)

### Cost of Memory Isolation with SURE

**SURE** has MPK enabled by default. To see the performance impact of MPK, we consider a variant of **SURE** with MPK disabled (`baseline`). We use the same functions, varying the message sizes.

***MPK in SURE has limited penalty.*** Fig. 4.10 shows the normalized latency and throughput. With a single connection, **SURE** shows 1.2-1.3× increased delay compared to the `baseline`. As we increase the number of concurrent connections, **SURE**'s RPS decreases (*e.g.,* 1.8× reduction at 64 connections). We believe this overhead is relatively small for the reward of robust memory-level isolation in our shared memory data plane and the unikernel TCB. `mprotect()` is a system call that can change the access privilege of specified memory pages, similar to MPK. However, as reported in [133], switching the access privilege with MPK incurs only ∼20 CPU cycles. Using `mprotect()` on the other hand requires more than 1000 CPU cycles to complete, resulting in much poorer performance.

### Improvement with library-based sidecar

We consider the widespread individual container sidecar as the baseline to compare with. Each sidecar connects to the user function container over the kernel loopback interface [2]. We use the NGINX proxy as the implementation of the sidecar, as demonstrated in production [145]. For the NGINX setup, we assign one CPU core to the NGINX sidecar and another core to the user function. In the **SURE** setup,

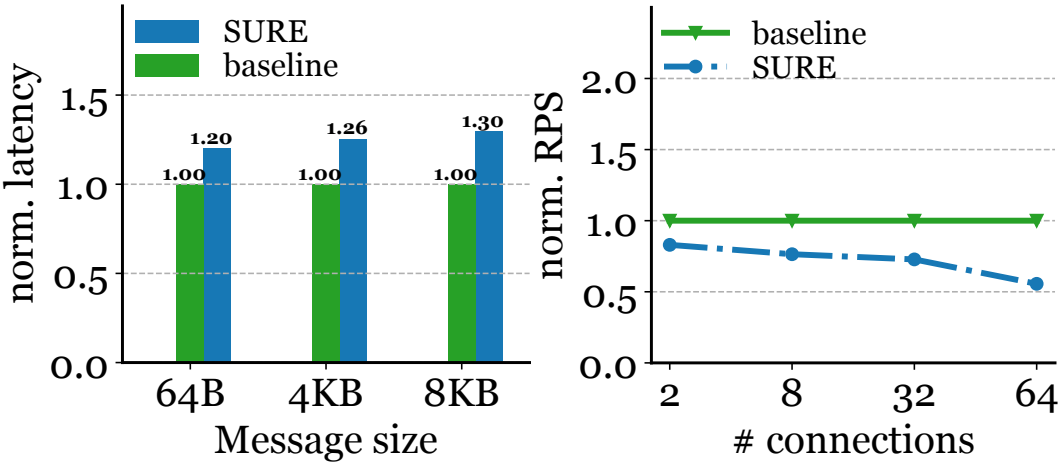


Fig. 4.10 The impact of MPK on **SURE**'s performance. We show the normalized latency and RPS.

| Msg size | **CPU cycles (× 1K)** | | **Added delay (us)** | | **Throughput (MBytes per second)** | | |
|---|---|---|---|---|---|---|---|
| | *SR* | *NGINX* | *SR* | *NGINX* | *SR (no SC)* | *SR* | *NGINX* |
| **256B** | 0.50 | 60.4 | 0.21 | 25.2 | 342 | 309 | 12.3 |
| **4KB** | 0.55 | 59.5 | 0.23 | 24.8 | 3697 | 3533 | 185 |
| **8KB** | 0.55 | 58.2 | 0.23 | 24.2 | 5525 | 5369 | 337 |

Table 4.1 Library-based sidecar (SR) vs. Individual sidecar (NGINX)

the library-based sidecar shares a CPU core with the user function code. Note that we measure the CPU cycle consumption and the added delay of the sidecar for a single connection. When measuring the throughput, we use concurrent connections to saturate the user function and sidecar. We show the throughput results with 64 connections in Table 4.1. However, the observations are consistent for other values (*e.g.,* 32, 128 connections).

***Library-based sidecar shows negligible overhead.*** Table 4.1 compares the CPU cycles spent on the sidecar for different alternatives. The CPU cycles consumed by our library-based sidecar are negligible compared to those of an NGINX sidecar (only 0.9%). The higher CPU cycle consumption by the NGINX sidecar is due to the loose coupling between the sidecar and the function, which results in additional overhead from the kernel's loopback interface. This extra CPU cycle consumption is also reflected in increased network delay and decreased throughput: the library-based sidecar adds only 0.21-0.23 $\mu$s to the data path, while the NGINX sidecar adds

more than 24$\mu$s of delay, potentially severely impacting data plane performance. The throughput of **SURE** with library-based sidecar is also close to **SURE** with sidecar disabled ("*SR (no SC)*" in Table 4.1). This shows the advantage of our library-based sidecar - maintain low latency and high throughput with negligible CPU consumption.

***Library-based sidecar has lower memory footprint.*** The individual sidecar serves as a reverse proxy between the user function and the external client, inevitably requiring additional dependencies and having a larger memory footprint. Our library-based sidecar avoids this overhead almost entirely. Our analysis shows that **SURE**'s library-based sidecar (125KB) reduces the memory footprint by 165×, compared to the NGINX sidecar (20.2MB). This brings many benefits, such as increased function density on every single node.

### Benefit of the zero-copy TCP/IP stack

We compare Z-stack (denoted `ZS`) with F-stack [18] (denoted `FS`) for protocol processing in the **SURE** gateway. The F-stack gateway incurs data copy when exchanging payloads with the function. We also compare Z-stack with Linux's kernel protocol stack (denoted `KS`) to evaluate the performance improvement and costs of using the DPDK PMD. In the `KS` setup, we let the function directly access kernel stack without involving the gateway. We use a TCP echo server/client pair (integrated with different alternatives) for this experiment. The server and the client are deployed on different nodes. `ZS` achieves a $\sim$1.2× RPS improvement and latency reduction under high traffic load (more than 100 connections) compared to `FS` (Fig. 4.11). This clearly showcases the advantage of having zero-copy protocol processing. On the other hand, `FS` inevitably introduces data copies between the server function and the F-stack gateway, resulting in lower performance. **SURE** also shows significant RPS improvement compared to the kernel protocol stack. **SURE** not only avoids data copies, it also eliminates other kernel-related overheads. Note that **SURE** shows slightly higher latency than `KS` under very light load with small packets (e.g., 64B, single connection in Fig. 4.11 (left)) as **SURE** uses the **SURE** gateway to relay between the function and Z-stack, resulting in some additional delay. But **SURE** is significantly better than `KS` for larger message sizes.

**Assessing the polling "tax" of Z-stack. SURE** chooses to use DPDK's busy-polling PMD to move packets between the **SURE** gateway (with the Z-stack) and the NIC.

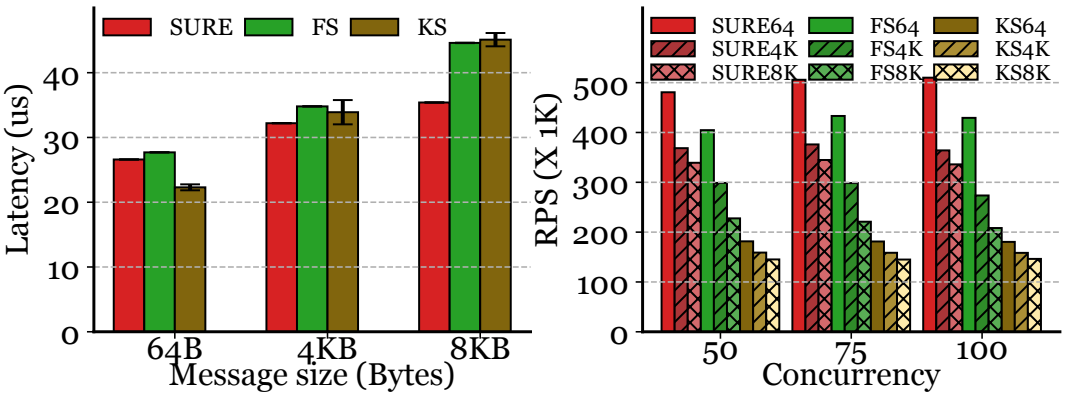


Fig. 4.11 Performance of inter-node data plane: (left) latency with 1 connection; (right) RPS under different concurrency levels. `FS`: F-stack; `KS`: kernel stack.

| Tested Load (X 1K RPS) | **Kernel stack CPU (%)** | | | **Z-stack CPU (%)** |
|---|---|---|---|---|
| | *Interrupt* | *Others* | *Total* | |
| ***30*** | 10 | 14 | **24** | 100 |
| ***60*** | 36 | 33 | **69** | 100 |
| ***90*** | 54 | 58 | 112 | **100** |
| ***120*** | 75 | 78 | 153 | **100** |
| ***240*** | 108 | 94 | 202 | **100** |

Table 4.2 Polling tax of Z-stack.

**SURE** dedicates a CPU core to the **SURE** gateway for protocol processing. This CPU cost is spent independent of traffic load, unlike an interrupt-driven kernel, and is the "polling tax" of Z-stack. The interrupt-driven kernel stack (`KS`) achieves better CPU efficiency at light load due to on-demand execution (at 30K, 60K RPS in Table 4.2). But, `KS` is inefficient for higher loads ($\geq$90K), because of interrupt handling [137] and other kernel overheads. In comparison, **SURE** achieves the same RPS consistently using a *single* CPU core with its kernel-bypass, eliminating interrupts. Busy-polling on a *single* CPU core has better overload behavior and is more efficient under heavy traffic load than an interrupt-based kernel stack. The increased function density on a single node (*e.g.,* with bin-packing-based function placement) can facilitate sharing of the **SURE** gateway and amortize the cost of busy polling.

### 4.6.2 Realistic Workload Evaluation

Online Boutique [143] is a microservices-based online store application from Google containing 10 functions and up to 6 different function chains in a serverful setup, by default using gRPC to interconnect functions (called "SF"). We use Knative as the baseline serverless platform to compare against (termed "SL"). We use Locust [146] as the load generator using the boutique's default workload [143]. Note, we disable the user wait time of the default workload to generate a heavier workload. We ported the boutique's microservices to **SURE**. We compare three alternatives: SF, SL, and **SURE**, with two distinct deployment settings: (1) All functions are deployed on the same node (intra-node), and (2) Frontend, Checkout, and Recommendation functions (3 intermediate functions that could become hotspots) are deployed on one node, the remaining (leaf) functions are deployed on a second node (inter-node).

**RPS and Tail Latency.** As shown in Fig. 4.12 (for intra-node setup), the RPS of **SURE** is up to 17× and 79× higher than the SF and SL. Both SF and SL are CPU resource limited (RPS doesn't increase) for concurrency above 16. In addition, the 95%ile latency of **SURE** (at a concurrency of 16) is below a millisecond (0.39ms), making **SURE** highly desirable for loosely-coupled, interactive microservices. This shows the compelling performance improvement from shared memory processing and library-based sidecar used by **SURE**.

With the inter-node setup (Fig. 4.13), **SURE** still maintains its superior performance compared to SF and SL: **SURE** achieves up to 6× and 19× higher RPS than SF and SL (concurrency of 8). **SURE**'s inter-node RPS is lower due to the protocol processing in the network gateway at the 2 nodes supporting the boutique functions (2.6× reduction). However, the zero-copy protocol processing in Z-stack still maintains a sub-millisecond latency, even at the 95%ile for **SURE** (CDF shown in Fig. 4.13 middle). SL's tail latency is 15× higher, and SF's tail latency is also 4.8× higher than **SURE**'s.

**CPU Consumption.** As concurrency goes up, the higher efficiency of **SURE** is clearly reflected in the dramatic scaling up of the RPS. This is despite the total CPU usage being much lower with higher concurrency levels. Even at a low concurrency where **SURE**'s CPU usage is higher (comes from polling and the use of a CPU for each function), **SURE** delivers a much higher RPS. In Fig. 4.12 (intra-node), as the concurrency level goes up, CPU usage of the other alternatives rapidly increases,

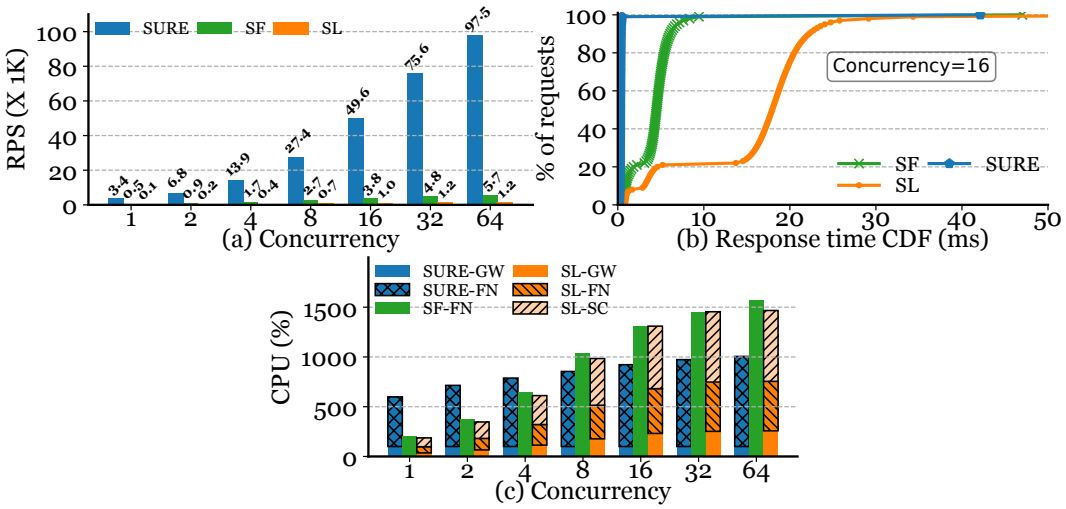


Fig. 4.12 Online boutique results (intra-node): (a) RPS, (b) Response Time, (c) CPU usage.

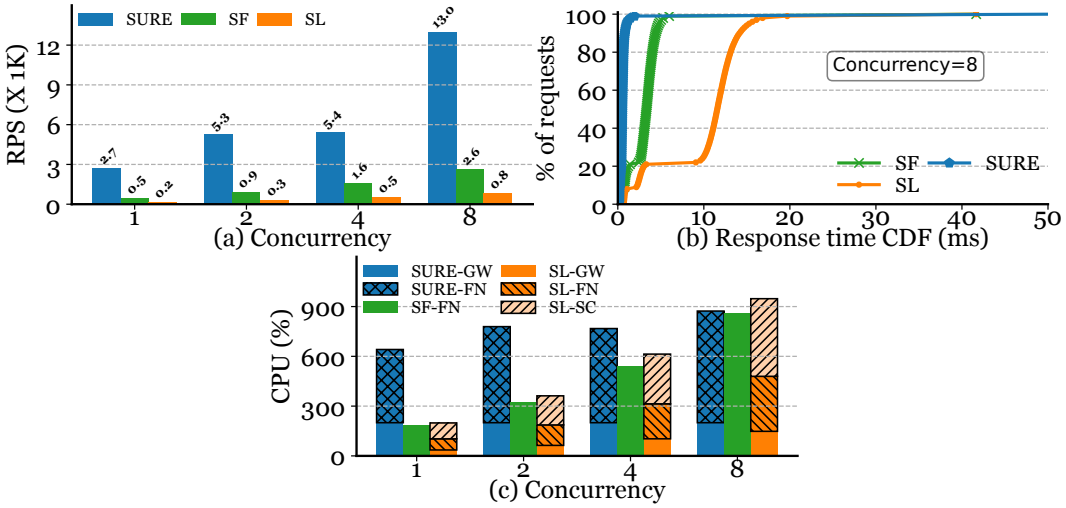


Fig. 4.13 Online boutique results (inter-node): (a) RPS, (b) Response Time, (c) CPU usage.

using up 15 CPU cores, compared to **SURE** using only up to 10 CPU cores at the maximum concurrency level. In Fig. 4.13 (inter-node), at the maximum concurrency level, all alternatives use 9 CPU cores (**SURE**'s RPS is much higher). The CPU usage of **SURE** gateway (`SURE-GW`) also does not grow, unlike `SL` (`SL-GW`). Moreover, `SL`'s sidecar uses up a very large amount of CPU (`SL-SC`), even more than the function itself at higher concurrency levels. All of this is also reflected in the response time behavior and RPS: `SL` is always worse than `SF` due to the heavyweight serverless components (individual sidecar and gateway). Note that our current implementation only supports allocating one CPU core to the **SURE** gateway, limiting its service

capacity (maximum concurrency level of 10). Our future work will vertically scale the **SURE** gateway to allocate more CPU cores to support higher concurrency.

## 4.7 Conclusions

**SURE** is a unikernel-based, lightweight serverless framework that offers high-performance inter-function networking and lightweight library-based sidecars for service mesh. It utilizes MPK and associated call gates and an enhanced unikernel TCB to mitigate the vulnerabilities of shared memory processing and single-address-space unikernels while retaining high performance and being efficient. Benchmarked against a serverful gRPC-based alternative in a multi-node deployment for a complex web workload, **SURE**'s data plane delivers up to 6× increase in throughput and a 4.8× reduction in tail latency while being more secure. The popular containerized serverless platform (Knative [147]) achieves only 5.3% of **SURE**'s throughput and has up to a 15× increase in tail latency.

# Part IV

# Improving the management of network services through eBPF

# Chapter 5

# Creating Disaggregated Network Services with eBPF: the Kubernetes Network Provider Use Case

## 5.1 Introduction

The extended Berkeley Packet Filter (eBPF) allows executing arbitrary code in different kernel hooks, which are triggered upon multiple events, such as a syscall or a packet received. This enables to extend the processing done by the kernel without changing its source code or loading kernel modules. eBPF code is sandboxed in order to guarantee that a user provided program can not compromise the functioning of the kernel. Several projects leverage eBPF to bring new functionality to the kernel in the fields of security, monitoring and networking. However, most of the networking-related projects implement new features as monolithic eBPF programs, making the code hard to maintain, to extend, and difficult to reuse in other use cases. In this respect, the Polycube software framework [32] addresses some of the well-known eBPF limitations when creating complex virtual network functions [40], and introduces the capability to split eBPF software in multiple, independent network functions, which can be arbitrarily connected in order to create a more complex service graph. This enables the creation of complex networking services by compositing elementary basic blocks (e.g., bridge, router, firewall, NAT, load balancer, and more), with an high degree of reusability.

Container networking is a perfect example of such scenario, and has gained key importance with the diffusion of the microservices architecture and the decomposition of applications in a collection of small, loosely-coupled services running in containers. Container orchestrators such as Kubernetes (K8s) must provide a flexible and efficient network infrastructure, since containers can be created and destroyed at high frequency, must exchange a lot of data and must be easily accessible from the Internet. However, K8s defines only a functional networking model and it relies on 3rd party plugins for the actual implementation of network services.

This chapter shows how a K8s network provider can be created using solely eBPF primitives according to the *disaggregated services* model, i.e., defining a modular architecture based on traditional network components such as routers, load balancers and NATs, and without giving up on performance.

This chapter is structured as follows. Section 5.2 provides an overview of the K8s networking model and how Polycube achieves service disaggregation. Section 5.3 describes the overall node architecture of our solution while section 5.4 shows a preliminary performance comparison compared to existing solutions. Finally, section 5.5 describes the relevant related work and section 5.6 draws our conclusions, highlighting the potential future work.

## 5.2 Background

### 5.2.1 Kubernetes networking

Kubernetes is an open-source orchestrator for containerized applications and defines a functional network architecture organized in three levels. (*i*) **Pods**, the basic scheduling concept, execute containerized applications that are connected to a default virtual network, whose outreach is limited to a single server (*node*, in the Kubernetes terminology). Pods are ephemeral entities that can be destroyed and re-spawned if needed, even on another node. Each server can host a maximum number of pods, usually organized in a contiguous and private address space (e.g., consecutive /24 networks on different nodes). (*ii*) **Physical nodes**, which inherit their addressing space from the physical datacenter network; for scalability reasons, this usually includes switched and routed portions. (*iii*) **Services**, a higher-level concept that enables the reachability of one or more (homogeneous) pods by means of the same

network identifier (e.g., IP address). This primitive guarantees the decoupling between the *service* IP endpoint, which remains stable, from the *actual* pod(s) that provides the service, whose IP can change (e.g., in case the pod is restarted in another location) or it may be present in multiple replicas (hence, multiple IPs could be used). Services leverage a third addressing space, disjoint from pod and datacenter addresses.

Kubernetes foresees three types of services. A **ClusterIP** identifies a service that is reachable only from pods within the cluster, or by an application that runs on the cluster nodes. A **NodePort** service, instead, is reachable from outside the datacenter: a TCP/UDP port `<nport>` is allocated to the service itself, and all packets directed to any `<node_ip_address:nport>` will be redirected to one of the pods associated to the given service. NodePorts are not widely used because they require (*i*) nodes with reachable (e.g., public) IP addresses; (*ii*) external users to know the IP addresses of the nodes and (even worse) (*iii*) the port that has been allocated. Finally, a **LoadBalancer** service allocates a public IP address associated to the given service, which is achieved by interacting with an external entity in charge of the above public addresses; all packets directed to the LoadBalancer IP address will be delivered to one of the corresponding pods.

Basic connectivity between pods is provided through the cooperation of data-center networking and plugins implementing the CNI (Container Network Interface) [148], a specification that enables changeable modules that configure network interfaces in Linux containers. The CNI specification is very simple, dealing only with network connectivity of containers and removing allocated resources when the container is deleted. Instead, packets to services are by default handled by a dedicated Kubernetes component, `kube-proxy`, which configures `iptables` with the proper rules to translate *service* IP addresses into *pod* IP addresses, and to implement load-balancing policies.

Most of the so-called CNI providers (e.g., Cilium, Calico, Flannel, etc) implement more than just the base CNI specification and include (*i*) data-center wide networking (either using an *overlay model*, e.g., through `vxlan` interfaces, or *direct routing*, hence interacting with the datacenter physical infrastructure to push the proper routes that satisfy K8s reachability rules) and (*ii*) IP address management (IPAM module). Instead, most of the CNI providers rely on `kube-proxy` for services: a packet coming from a pod and directed to a service is delivered by the CNI to kube-proxy,

which operates the proper transformation on IP addresses and ports and returns it again to the network plug-in, which takes care of delivering it to the target pod, possibly traversing the datacenter network.

K8s adopts a functional model for networking, defining a set of behavioral rules for network connectivity[1] but without specifying how those must be implemented by the specific network provider. In addition to rules already mentioned for services, it adds the following ones for basic connectivity: (*i*) all pods can communicate with all other pods without NAT; (*ii*) agents on a node can communicate with all pods on that node; (*iii*) pods in the host network of a node can communicate with all pods on all nodes without NAT. It turns out that network providers have full freedom to choose their own implementation strategy, hence privileging e.g., the easiness of use, performance, scalability, and more.

However, this freedom is widely recognized as a nightmare, being very difficult to understand how each network provider works under the hood, hence severely impairing the capability of a network engineer to debug a problem. This is exacerbated by the complexity of the Kubernetes networking, which overall includes functions such as bridging and routing (for pod-to-pod connectivity), load balancing and NAT (for pod-to-service and Internet-to-service), and masquerading NAT (for pod-to-Internet), not to mention the necessity of security policies (hence, firewalls) to protect both pod-to-everything and external-to-everything communications.

Finally, all the above networking components must be integrated with a control plane, which interact with K8s and detect any change in the status of the cluster (e.g., a node/pod/service is added/removed, a service is scaled up/down, etc.), hence propagating the required configurations in all involved components (e.g., adding a new node requires configuring a new route toward that node in the routing table of all existing nodes).

Given this complexity, it becomes evident that the capability to rely on well-known (disaggregated) functions (e.g., bridging, routing, load balancers), each one running with their configurations and state, would greatly simplify day-by-day monitoring and debugging operations compared to network providers created according to the monolithic model.

[1] https://kubernetes.io/docs/concepts/services-networking/.

### 5.2.2 Service disaggregation with Polycube

Polycube [32] is an open-source software framework based on eBPF that enables the creation of arbitrary network function chains, adopting the same model (boxes connected through wires) currently used in the physical world. Polycube network functions, called *cubes*, are composed by an eBPF-based data plane running in kernel (actually one or more eBPF programs) and a control/management plane running in user space. A user space daemon (*polycubed*) provides a centralized point of control, allowing to access the configuration and state of cubes through a RESTful API. Cubes can be seamlessly connected to each other or to network interfaces through *virtual ports*, an abstraction that is implemented through an eBPF wrapping program that performs some pre- and post- processing in order to receive/send packets from the previous/to the next component in the chain. To implement this redirection mechanism, each port is identified by a unique ID inside the cube, and each cube maintains a forward chain map containing information about the peer associated to each port. This information is used by the post-processor to apply the correct action to forward the packet, that could be either a *tail call* to the eBPF data plane of the next cube or a `bpf_redirect()` to a destination interface. Figure 5.1 shows an example of this mechanism for a simple topology composed of one router and one bridge. Chaining capabilities of Polycube represent a suitable way to create disaggregated services; however, this solution has never been validated in a complex scenario such as K8s networking.

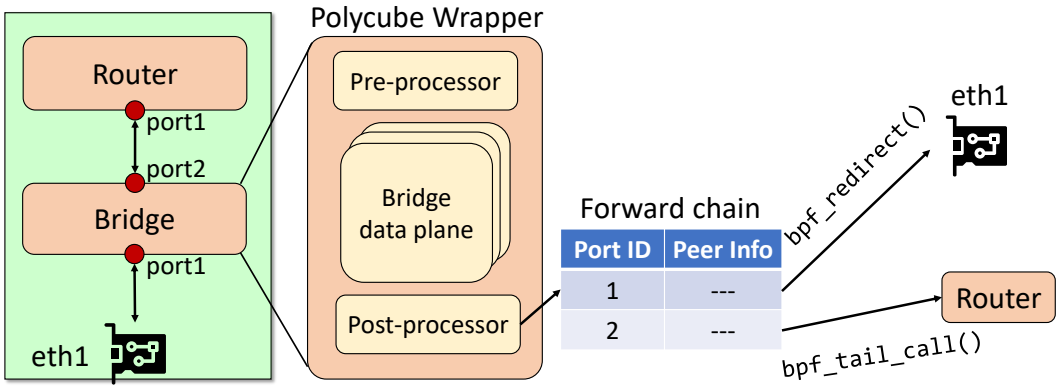


Fig. 5.1 Chaining and ports in Polycube.

## 5.3 Architecture

Our proposed architecture targets the entire K8s networking, including also services and, potentially, network policies, hence replacing also `kube-proxy`, i.e., the component that provides cluster-wide service-to-pod translation and load balancing. Our network provider supports ClusterIP and NodePort services and relies on a VxLAN overlay network to support inter-node communication. The current version of the prototype does not support security policies but, thanks to the modular approach, this and other functionalities can easily be added without any change in the other components. The architecture (shown in fig. 5.2) leverages four Polycube-based independent network services, while it relies on the kernel to handle the lifecycle of VxLAN tunnels.

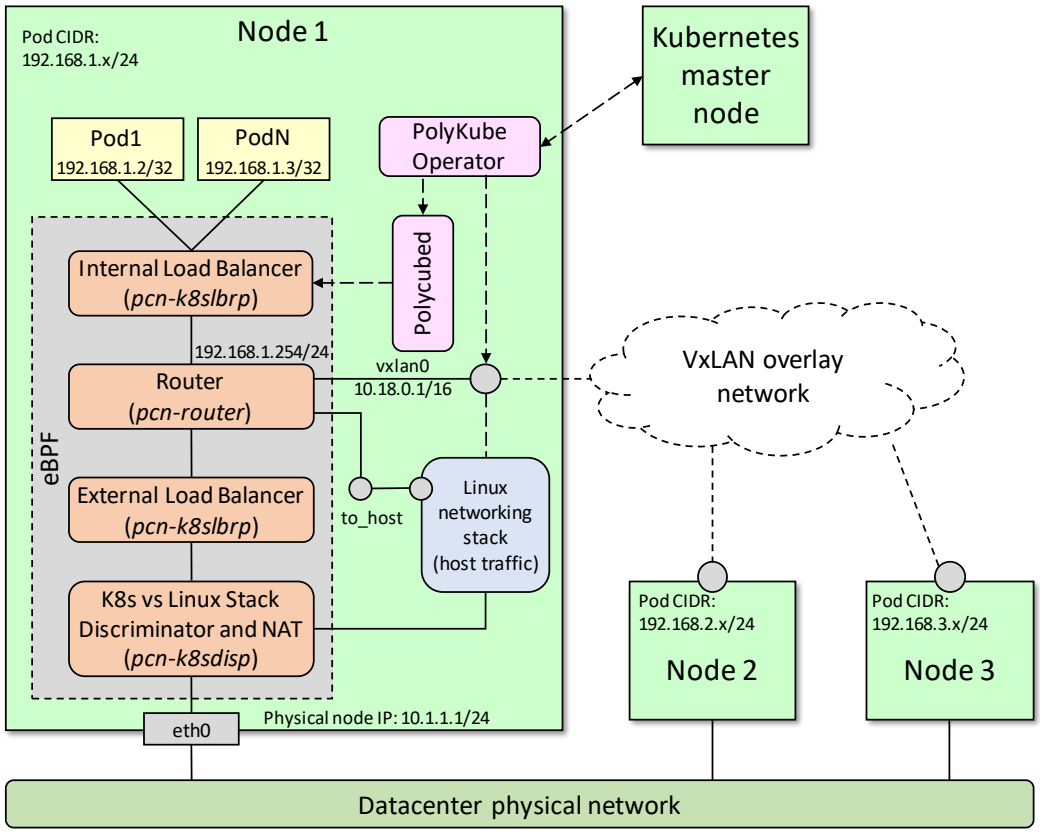


Fig. 5.2 Overall architecture of the eBPF K8s network provider.

### 5.3.1 Main components

**K8s vs Linux Stack Discriminator and NAT (DISC-NAT).** This service performs source NATting for the pod-to-Internet traffic, replacing the address of the pod with the address of the node. For incoming packets, it distinguishes among traffic directed to the host (either directly or because it needs VxLAN processing) and traffic directed to pods (an external host trying to contact a NodePort service or the return traffic of a pod). This is done by checking that the packet (*i*) does not belong to a NATted session (i.e., a lookup in the NAT session table of the service), and (*ii*) that is not directed to a NodePort service (i.e., a check against the TCP/UDP destination port of the packet). In case both lookups fail, the packet is sent to the Linux network stack. Vice versa, in the first case we apply the reverse NAT rule and the packet continues its journey towards the pods. In the second case, different actions can be applied according to the *ExternalTrafficPolicy* of the rule. If the policy is *local*, the traffic is allowed to reach only backend pods located on the current node, hence the packet can proceed towards the pod without modifications. In case the policy is *cluster*, the packet can also reach backend pods located on other nodes. Since later in the chain the packet will be processed by a load balancer and we must guarantee that the return packet will transit through the same load balancer, we apply source NAT replacing the source IP address with the address of a fictitious pod belonging to the PodCIDR of the current node (currently the first address of the range is used).

**External Load Balancer (ELB).** This element maps new sessions coming from the Internet and directed to a NodePort service to a corresponding backend based on the 5-tuple of the first packet. This load balancing decision is stored in a session table, implemented as a Least Recently Used (LRU) eBPF map, and reused for all subsequent packets. Old sessions are automatically purged by the LRU map. Incoming packets are updated with destination/port of the backend, while source fields are restored to service values for outgoing traffic.

**Router.** Since K8s requires that (*i*) all the pods in the cluster can communicate with all pods without NAT and (*ii*) different network addresses (PodCIDR) are allocated to pods on each node, a routing component is required. To facilitate the operations of the Internal Load Balancer (see later), we do not use a bridge between pods, hence forcing all pod traffic to be always delivered to the router. In fact, our network plugin assigns a /32 network to each pod and adds an ARP static entry for the gateway; hence all the packets are sent directly to the gateway, with pods never issuing any

ARP request. The router is configured with four ports: (1) towards the physical interface of the node; (2) towards local pods, configured with the proper PodCIDR (e.g., /24); (3) to enable the reachability of K8s processes and pods running in the host network (e.g., kubelet); (4) connected to a kernel VxLAN interface, which is used for inter-node pod-to-pod communication.

**Internal Load Balancer (ILB).** This load balancer operates on traffic coming from local pods and directed to ClusterIP services; all the other traffic is forwarded as is. This module has two types of ports; 'edge' ports are connected to entities that generate new sessions (hence, pods), while the 'server' port is the one used to reach the final servers, hence is connected to the router. Edge ports are configured with the IP address of the pod in order to be able to forward packets coming from the router to the correct destination. The load balancing logic is the same of the ELB, with a service-specific *InternalTrafficPolicy* attribute determining which backends (*local* or *cluster*-wide) are configured in the load balancer.

**K8s Control Logic.** A K8s operator is in charge of reacting to the cluster events and reconfigure the required network parameters in the controlled cluster. The operator has to react to events related to the following three Kubernetes resources.

(1) *Nodes*: when a node joins/leaves the cluster, a route is added/removed in the *Router* to update the reachability of the given PodCIDR through the VxLAN overlay network, and the node address is added to the VxLAN configuration.

(2) *Services*: the operator watches events regarding ClusterIP and NodePort services. ClusterIP services trigger an update of the ILB, while a NodePort triggers an update of the ELB and DISC-NAT module for the obvious reasons, as well as the ILB because of the creation of the ClusterIP address that is associated with the NodePort service.

(3) *Endpoints*: the operator watches any event that refers to Endpoints associated to Services. For each pair (address, port) extracted from the endpoints object, the corresponding service backend is updated on the proper Load Balancer: the ILB for ClusterIP services and both for NodePort services.

This component is deployed as a K8s DaemonSet, which ensures it runs on any node; K8s adopts a distributed configuration, in which each node has its own agent in charge of the node network configuration. This DaemonSet runs as privileged pod, and it includes the `polykube-operator` container (running the actual K8s

control plane) and the `polycubed` container (running the Polycube daemon). The two communicate through the node loopback interface.

### 5.3.2 Communication scenarios

The main communication scenarios of a typical K8s cluster, as shown in fig. 5.3, are implemented as follows.

**Pod-to-pod.** The originating pod sends its packets to the ILB, which transparently forwards them to the Router. If the destination is on the same node, the traffic is sent back immediately and the ILB forwards it to the destination. If the destination pod is on another node, the router redirects the packets to the VxLAN interface, where they are encapsulated by the kernel and forwarded to the destination node. On the remote node, the DISC-NAT passes the traffic to the kernel, which decapsulates it and sends it to the router and then to destination pod.

**Pod-to-Internet.** The traffic traverses the ILB, then the router forwards it towards the physical interface of the node, hence transparently crossing also the ELB. The NAT, instead, applies a source NATting rule, hence replacing the address of the pod with the one of the node, as well as the source port with a new available one. This allows the return traffic to reach the correct node without the necessity to advertise the PodCIDR on the external network. On the return path, the NAT checks if the packet belongs to a translated session; if so, it replaces the destination address and port of incoming packets with the ones of the original pod.

**Pod-to-service.** The pod traffic toward a ClusterIP service is first processed by the ILB, which selects a proper backend pod and updates the destination address and port of packets. Traffic is then handled by the router in the same way as with the pod-to-pod communication. For return traffic, the ILB checks if the packet belongs to a translated session; if so, it restores the original source service address and port.

**Internet-to-service.** When a remote host wants to contact a NodePort service, the packet is first processed by the DISC-NAT, that identifies it as targeting a NodePort service (based on the destination port). If the *ExternalTrafficPolicy* of the service is *cluster*, the DISC-NAT updates the source address of the packet with the one of the fictitious pod, to guarantee that the return packet will come to the same node. The packet is then forwarded to the ELB, which (*i*) selects a proper backend pod, (*ii*) updates the destination address and port with the ones of the backend, and (*iii*)

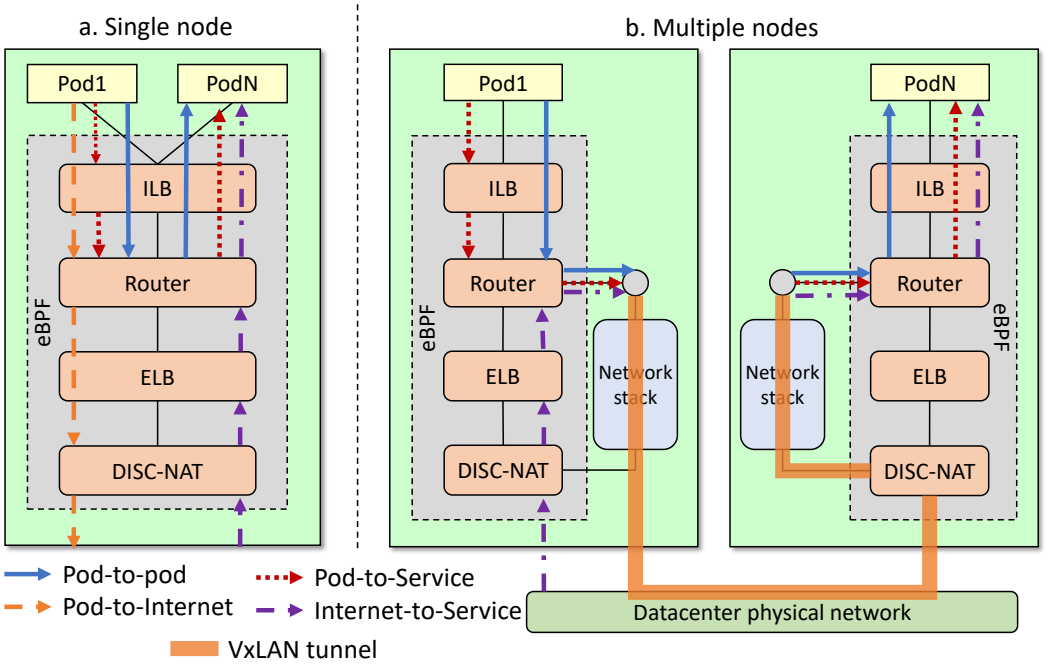


Fig. 5.3 Modules involved in single (a) and multi-node (b) communications.

forwards the packet to the router, that can handle it such as in normal pod-to-pod communications.

## 5.4 Evaluation

This section presents an assessment of the performance provided by our network provider under different circumstances, to determine whether the disaggregated approach would introduce any noticeable performance penalty. We run the tests using the `iperf3` tool, configured with the default parameters; the server was always running in a pod, while the client was either in a physical machine or in another pod depending on the test. Tests were carried out on a cluster of 2 nodes, each one featuring a CPU Intel® Xeon® CPU E3-1245v5@3.50GHz (4 cores plus HyperThreading), 64GB RAM, and a dualport 40 GbE Ethernet XL710 QSFP+ card, all running Linux kernel v5.4.0 and K8s v1.23.5. Tests involve other two eBPF-based solutions (namely, Cilium and Calico) and a widely used 'traditional' approach such as **Flannel** [149]. All providers were deployed using the VxLAN overlay model; Cilium and Calico were configured with their eBPF kube-proxy replacement, hence enabling a complete eBPF data plane such as in our solution.

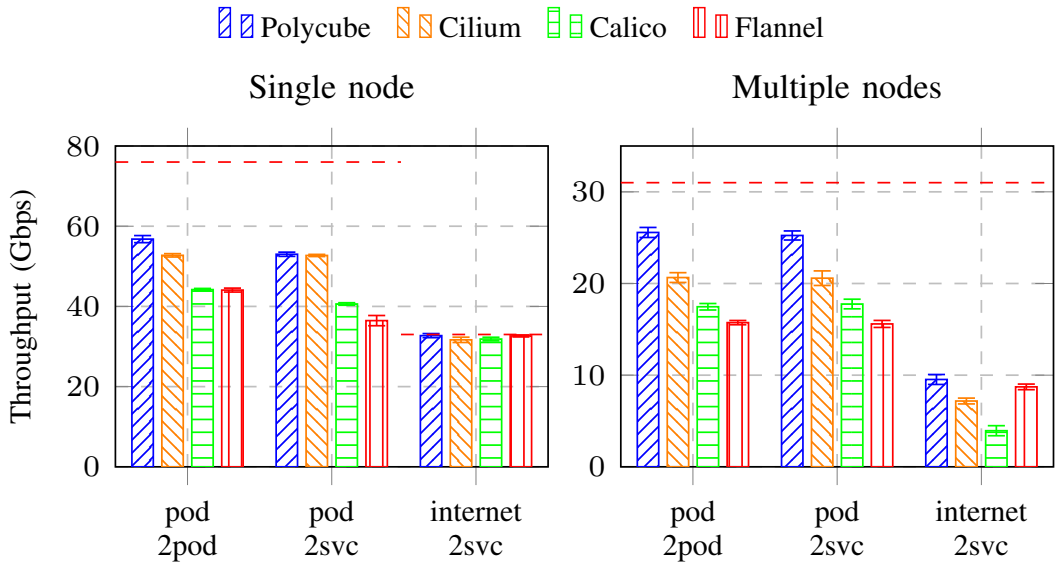


Fig. 5.4 Throughput comparison (TCP).

We considered the following communication scenarios: **pod-to-pod**: a pod client connects to the actual IP address of a pod server, showing the performance of the base networking without load balancing; **pod-to-service**: a pod client connects to a pod server using its ClusterIP service, to evaluate the performance of the load balancer as well as the L3 routing; **internet-to-service**: the client is executed in an external host and the pod server is accessed through its NodePort service. Tests were performed with pods running both on a single node and on multiple nodes, with the latter adding the overhead of the VxLAN encapsulation and the limitation of the physical network (link speed, PCI bus) and requiring to cross the physical network twice ((*i*) client to receiving node; (*ii*) receiving node to backend node) for the *internet-to-service* tests. Results are depicted in fig. 5.4, with the red dashed lines representing the baseline achieved running bare metal `iperf3` on localhost (in case of single node) or between two nodes. As expected, the throughput decreases when the traffic traverses a larger number of network components. However, despite the disaggregated architecture, our solution provides always better performance compared to other solutions, with even higher margins when considering multiple nodes. While this result may be impacted by other providers supporting more features than our PoC code, such as network policies, these have not been used in the tests, hence providing the ground for a fair comparison. Overall, this suggests how disaggregation does not introduce performance penalties compared to a traditional monolithic approach.

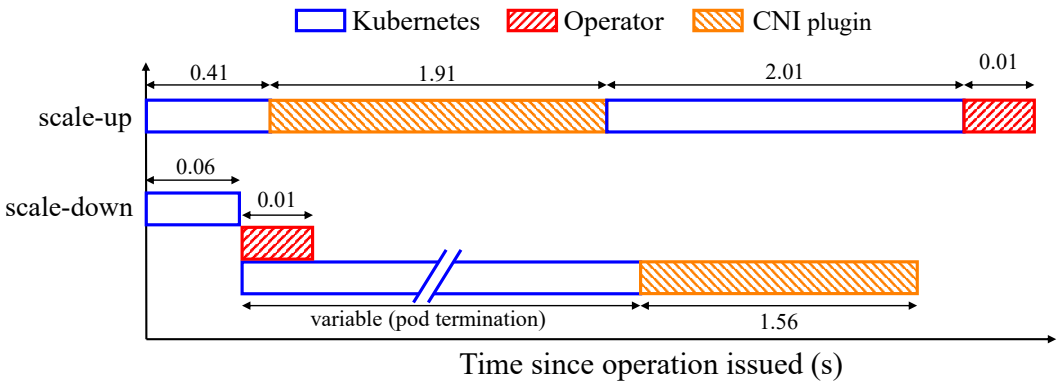


Fig. 5.5 Reaction time in case of scale up/down events.

Figure 5.5 shows the reaction time of our operator and CNI plugin when requiring to scale up/down the pods of a service, compared with time required by other Kubernetes components until connectivity to the target pod is available/disabled. Results show that the time taken by our components is negligible compared to the overall time required by K8s to react.

## 5.5 Related work

Among the many eBPF-based network services, we cite here only the ones that are most representative in this space.

**Katran** [23] represents a software solution to offer scalable network load balancing to layer 4 that leverages eBPF/XDP to provide fast packet processing. While being very sophisticated, it has been engineered to be the sole (monolithic) network function active on the network path, hence preventing the deployment of other functions operating on the same traffic.

**Cilium** [110] provides networking, security and observability for cloud-native environments such as Kubernetes clusters. Cilium is based on eBPF, which allows for the dynamic insertion of powerful network security, visibility and control logic into the Linux kernel. In Cilium, eBPF is used to provide high-performance networking, multi-cluster and multi-cloud capabilities, advanced load balancing, transparent encryption, extended network security features, and much more. While providing observability primitives through its *Hubble* module, its internals are rather complex and made with a monolithic approach. Cilium defines a set of six logical objects

(Prefilter, Endpoint Policy, Service, etc.), based on six different features offered by the provider (DoS mitigation, network policies, load balancing, etc.). However, these objects are not mapped into clearly separated modules, neither for the data plane (their logic is scattered among different intertwined eBPF programs), nor from a topological point of view (cannot track the path of a packet or capture the traffic flowing from one object to another), nor from a control plane perspective (cannot configure and inspect these objects independently). Similar characteristics can be found in **Calico** [150], which recently adopted the eBPF/XDP technology as well.

## 5.6 Conclusions

We presented a network provider for Kubernetes based on disaggregated eBPF services, which improves monitoring and debugging as well as how code can be maintained, extended and reused. Our open-source solution[2] demonstrates the feasibility of the disaggregated approach in eBPF and our preliminary evaluation shows no particular overhead introduced by our model with respect to another state-of-the art monolithic solution. As a future work we plan to introduce support for (*i*) direct routing and (*ii*) network policies.

[2]Code and docs available at https://github.com/polycube-network/polykube.

# Chapter 6

# Enabling Scalable SFCs in Kubernetes with eBPF-based Cross-Connections

## 6.1 Introduction

With the ongoing softwarization of the network and the advent of Network Function Virtualization (NFV), Network Functions (NFs) have been increasingly deployed as software rather than hardware appliances. The recent trend towards the increasing use of microservices instead of huge, monolithic functions, led to the necessity to define a way to create chains of NFs, forming Service Function Chains (SFCs), which are characterized by an ordered set of NFs traversed by the traffic in a precise sequence. Since the amount of traffic traversing the chain is highly variable, SFCs must be able to dynamically adapt to the current network traffic load, hence optimizing the use of resources and improving the quality of the offered service. With the advent of cloud-native NFs [151], new possibilities have opened up for the implementation of more flexible SFCs that can benefit from cloud-native environments. In particular, cloud-native infrastructures such as Kubernetes include the logic to provide automatic scalability of the running services within their core functionalities. However, although Kubernetes is currently the de facto standard orchestrator for general-purpose applications, it lacks some functionalities required by NF workloads, such as the possibility of defining service chains with precise

network topologies and configurations, and the possibility to have pods with multiple network interfaces. Furthermore, the *Service* abstraction provided by Kubernetes, which leverages horizontal scaling and provides load balancing for applications, is not suitable for NFs since typically they are not the final recipient of the network traffic.

To address the aforementioned limitations of Kubernetes while bringing several of its advanced features also to the world of SFC services, we propose a model that integrates SFCs in Kubernetes, with the explicit goal of maximizing the reuse of existing Kubernetes features (in particular, horizontal pod autoscaling), while current SFC technologies tend to propose dedicated mechanisms for SFCs. This simplifies the creation of cloud-native SFCs and, with respect to automated scaling, it leads to a higher efficiency thanks to the capability to dynamically adapt to the actual traffic load. In fact, the proposed solution enables each NF to be independently scaled in an arbitrary number of replicas, hence providing flexible cross-connections between adjacent NFs instances in the chain.

This chapter is structured as follows. Section 6.2 presents the current state-of-the-art. Section 6.3 details our model for a scalable cloud-native SFC, while Section 6.4 provides a brief insight of a proof-of-concept implementation. Finally, the experimental evaluation is presented in Section 6.5, while Section 6.6 concludes the chapter.

## 6.2 Related Work

Several approaches for the provisioning of SFCs based on Software Defined Networking (SDN) and NFV technologies can be found in the literature, which has been comprehensively analyzed in [152]. Recently, the growth of cloud-native technologies for NFs and the success of Kubernetes as orchestration platform paved the way towards new techniques for SFCs provisioning, such as in [153] and [154].

A solution based on the Network Service Mesh (NSM) framework [155] was proposed in [153]. NSM allows individual workloads to securely connect to Network Services, independently of where they are running. A Network Service can be composed of a chain of Endpoints, which actually implement the NFs. NSM provides and manages all the necessary interconnections mechanism to let the traffic pass

from the client workloads to the endpoints of the requested Network Services. When a client workload requests a particular Network Service, NSM creates the necessary interfaces on the client and on the endpoints, and configures the underneath forwarding mechanisms in order to steer the traffic across the chain. However, NSM does not enable any efficient load balancing among NF Endpoint replicas. For this reason, the authors included a network-aware load balancing system in order to leverage all the instances of a particular Endpoint.

A framework based on Contiv/VPP [156] that integrates SFCs in Kubernetes was proposed in [154]. It consists in a Kubernetes Container Network Interface (CNI) plugin that uses FD.io VPP to provide network connectivity between pods. Contiv-VPP supports pods with multiple custom interfaces and enables chaining between pods. Since Contiv-VPP enables only SFCs that are composed of single NFs instances, scaling requires the replication of the whole chain, with a further load balancer that distributes the traffic among the different paths.

Unlike these two solutions that require either (*i*) the creation of new interfaces and links for each client session or (*ii*) the replication of the whole SFC, our solution allows to define NF cross-connections only once, when the chain is initialized, and leverages the native Kubernetes autoscaling at the NF-level granularity, improving the efficiency and simplicity the solution. This was achieved introducing the new concept of flexible-cross connections, which connect replicas of adjacent NF and are dynamically adapted in case of scaling events.

## 6.3 System design

### 6.3.1 Goals

Since Kubernetes does not provide any abstraction to support SFCs, our solution (*i*) introduces a proper model for SFCs, and (*ii*) defines the logic required to support multiple and independent NFs replicas in the chains, while leveraging existing Kubernetes features to the maximum extent. In details, our work addresses the following four goals.

*Declarative definition of SFCs.* SFCs must be created with a simple declarative description that includes only (logical) NFs and their interconnecting links, without

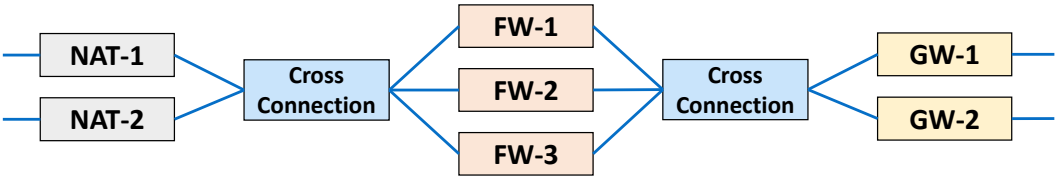


Fig. 6.1 An example of a scalable SFC composed of a NAT, a Firewall and a Gateway.

any further low-level detail, such as the number of NF replicas in each stage of the chain, or how the traffic is distributed among existing replicas.

*Automatic support for multiple replicas of a NF*. Each NF in the chain may be executed with an arbitrary number of replicas (e.g., Figure 6.1). This enables the creation of multiple paths across NFs of the chain for optimal workload distribution, which demands for a clever traffic distribution mechanism that is implemented transparently by our solution.

*Automatic adaptation of the chain to the variation of replicas*. Each NF in the chain must be able to scale (e.g., in/out) automatically, mostly leveraging existing Kubernetes mechanisms (e.g., horizontal autoscaling). This requires our solution to dynamically adapt the cross-connections between consecutive NFs to keep them aligned with the number of instances present at any given moment. This ensures that new replicas are used and no traffic is forwarded to deleted replicas, which can be achieved through consistent hash mechanisms to keep established sessions to the existing replicas.

*Support for L2/L3 NFs*. The solution must support both Layer 3 NFs, which have a MAC/IP address on their interfaces (e.g., a router), and NFs operating as bump-in-the-wire transparent middleboxes, processing all traffic flowing through them regardless of L2 addressing (e.g., a transparent firewall).

This work leverages existing Kubernetes mechanisms to decide when and how to scale NF replicas; hence, how to allocate resources and where to schedule each pod. However, pure standard Kubernetes algorithms might not always take the optimal decision for SFC workloads; as illustrative examples, the default autoscaling criteria is based on CPU load, which is not appropriate for NFs running according to the busy-polling model; the default scheduling policy may not recognize the necessity to run two consecutive NFs on the same server to minimize I/O traffic, and more. The solution consists in creating additional Kubernetes components that, for example, export new metrics to drive the autoscaling mechanisms (e.g., based on the amount

of traffic instead of the CPU used), or additional policies (e.g., affinities) to influence the decisions of the scheduler. Nevertheless, our approach enables reusing a large amount of tested code already present in Kubernetes, which greatly reduces the number of problems that need to be considered when running a SFC.

### 6.3.2 Modeling SFCs

We modeled a SFC as a list of nodes, each one specifying a NF, and a logical connection for each data plane interface of the NF (Figure 6.2a) that defines the next NF that can be reached through the interface. Each NF (Figure 6.2b) is characterized by one or more data plane interfaces, each one with its own network configuration (e.g., MAC/IP addresses, etc.; Figure 6.2c). One additional interface could be natively handled by Kubernetes and attached to the main CNI plug-in, which can be used for management purposes. Since a running NF can include one or more replicas, interfaces of each replica have exactly the same network configuration (including IP/MAC address, if present), hence making each replica just a new pool of computing resources that is (from the data plane point of view) indistinguishable from other ones.

This high-level connection model enables also the support for asymmetric NFs, e.g., a NAT, which apply a different processing to traffic according to its direction.

#### Modeling flexible cross-connections

To enable the connectivity between multiple replicas of adjacent NFs, we need a flexible *cross-connection* that connects adjacent groups of NF replicas according to the corresponding logical connections specified in the SFC models. Each instance of the cross-connection is two-sided and bidirectional, i.e., it can only manage traffic between two consecutive NFs, with packets that can flow in both directions. The number of handled replicas may be asymmetrical on the two sides and also variable over time.

Flexible cross-connections, compared to simple point-to-point links, have to manage a higher level of complexity deriving from the larger number of possible connections between NF groups. Furthermore, they provide traffic forwarding capabilities and all the necessary logic to distribute the traffic among all the NF

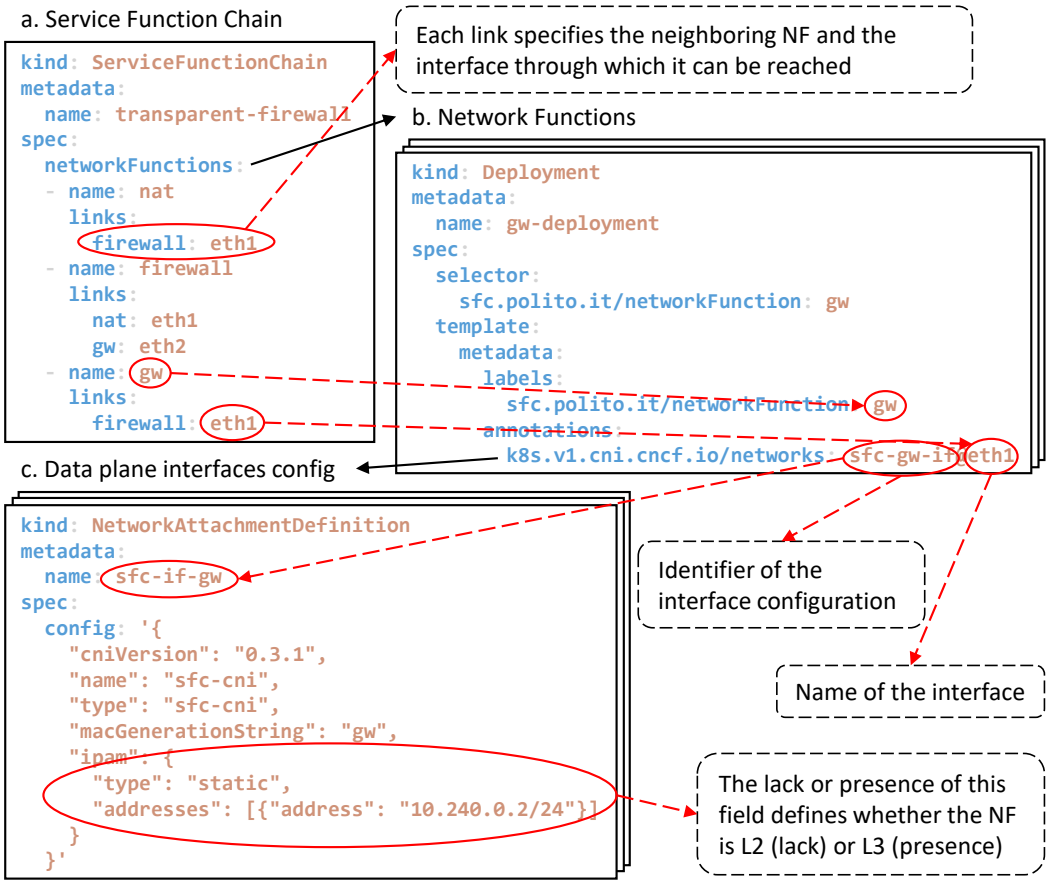


Fig. 6.2 Description of a simple SFC, referred to the chain of Figure 6.1.

replicas. The logic that determines how the traffic is distributed is based on network sessions (e.g., TCP/UDP 5-tuple), coupled with consistent hashing mechanisms to ensure that all the packets belonging to the same sessions are always forwarded to the same NF instances. This association between session flows and NF replicas occurs for each step of the SFC. As a result, the traffic of each session traverses a path composed of exactly one replica for each NF. All the above characteristics are transparent to NFs, which are not aware of the presence of any intermediate cross-connection, nor have any knowledge about the preceding/following NF replicas.

## 6.4 Implementation Overview

This section presents an open-source[1] PoC implementation of our model. Since the system was conceived to be integrated with Kubernetes, some of the prototype components have been designed to leverage the extensibility features of the aforementioned platform. This is the case of the *SFC CNI plugin*, which allows to configure the data plane interfaces of NF pods, and of the *SFC Operator* which acts as a manager for SFCs resources in the cluster, allowing to instantiate chains given their logical model. The other fundamental component is the *eBPF Load Balancer*, which implements the flexible cross-connections between NFs.

### 6.4.1 SFC CNI Plugin

The SFC CNI plugin allows to configure the L2/L3 data plane interfaces on NF pods, and has been designed to operate in conjunction with Multus CNI [157]. Multus is a meta-plugin that allows the use of multiple CNI plugins, to support multiple NICs in Kubernetes. By default, Kubernetes allows to have only one NIC for each pod, which is connected to the main cluster network. However, one NIC is usually not enough for NFs, since they typically need at least two additional interfaces (ingress and egress).

The SFC CNI plugin configures `veth` pairs that connect the pods with the network namespace of the host on which they are scheduled. For what concerns IP addressing, the system requires the use of the *static* IPAM plugin in order to assign the same addresses to all the replicas of a NF. Similarly, also the MAC address assigned by this plugin is required to be the same for groups of replicas. In addition, the SFC CNI plugin stores the details about interfaces configuration on the corresponding Kubernetes pod resource, in order to make this information available to the SFC Operator when it has to connect the pod to the eBPF-based load balancers.

### 6.4.2 eBPF Load Balancer

Our flexible cross-connections are based on eBPF [19], which has been chosen for its proven capability to create efficient network functions [40]; in fact, it can process

[1] https://github.com/fmonaco96/sfc-k8s

the traffic on its natural path inside the Linux network stack, with clear benefits in terms of performance and transparency. The eBPF load balancer handles the traffic between two NFs, irrespective of the actual number of replicas. Hence, it can implement an N-to-M cross-connection between all `veth` interfaces associated to two consecutive NFs in the SFC, which terminate in the *host* network namespace.

The eBPF load balancer operates at the Traffic Control (TC) level in the Linux networking stack, leveraging the hook points of the `veth` on the host side. This allows the packets coming from the pods to be immediately processed as soon as they arrive at the ingress interface of the host, hence without touching in any way the container where the NF is running. We rely on some internal eBPF maps to keep all the information about connected NFs and handled sessions. In particular, they leverage two array maps containing the indexes of the NF interfaces connected on each side. The session table, which is a hash table, associates each handled TCP/UDP 5-tuple with two interface indexes: one is the index of the interface from which the first packet of the session was received and the other one is the index of the egress interface selected by the load balancing logic for that session.

This selection is made in two phases: (*i*) an hash function is applied to the session 5-tuple and (*ii*) the obtained value is used as an index to select an entry from the array map of the egress interface indexes. A more elaborate consistent hashing mechanism could be used to limit the shuffling of sessions that can not be stored in the session table when the latter is full, but this optimization is left as a future work. Furthermore, an additional hash map is used as a small ARP cache to facilitate the management of the ARP protocol across the chain. In fact, since the IP/MAC addressing is the same for each group of replicas, the ARP request/replies are always identical. Responses can therefore be cached so that, after the first reply, load balancers are able to respond immediately with the cached reply, without having to propagate the request across the chain.

### 6.4.3 SFC Operator

The SFC Operator acts as a manager for the system implementing the *Operator Pattern* of Kubernetes. It operates as a controller for `ServiceFunctionChain` and `LoadBalancer` custom resources, which represent the SFCs and eBPF Load Balancers instances in a cluster and are stored in `etcd`. This operator runs as

privileged DaemonSet in the hosts' network namespace, due to the necessity to manipulate the network stack and attach the eBPF Load Balancers to `veth` interfaces. Moreover, it watches the Kubernetes resource representing the NF pods in order to immediately detect if there is a new replica or an existing one is going to be deleted.

The SFC Operator has been implemented using Kopf [158], a framework that allows to write Kubernetes Operators in Python. This allowed us to leverage the BCC toolkit [159] to handle the eBPF Load Balancer instances. In this way, it was possible to create a single program that could deal with the management of events concerning SFC cluster resources, but also with the practical aspects concerning the configuration of the data plane of eBPF Load Balancers. When the SFC Operator notices that there is a new SFC resource in the cluster, it creates the load balancers across the chain, compiling and injecting the corresponding eBPF code in the Linux kernel. It then configures the TC hook of the NFs interfaces so that all the generated traffic is handled by the load balancing logic, providing a logical link between NFs and load balancers. During the operation of the chain, when it detects an event on NF replicas, it updates the configuration of adjacent eBPF Load Balancers so that they are always aligned to work with the instances present at the moment.

### 6.4.4 SFC Lifecycle

The lifecycle of a SFC is determined by its corresponding `ServiceFunctionChain` resource. These resources allow users to create SFCs in a fully declarative way without the need of any other further detail, leaving the practical operations to the SFC Operator. Figure 6.3 shows the main actions performed by the SFC Operator when it has to manage the creation of a new chain. As soon as a user applies a manifest of a SFC to the Kubernetes API Server, (1) the SFC operator watches the related event and (2) parses the provided description in order to obtain the chain structure. During this phase, for each couple of adjacent NFs, it creates and pushes to the API server a `LoadBalancer` manifest and, at the same time, through another controller, it proceeds with the actual instantiation of the load balancers injecting their eBPF code in the kernel (3). After their successful creation, the SFC Operator proceeds with the creation of the links between NF pods `veth` interfaces and adjacent load balancers (4), configuring the TC hook of the pods `veth` interfaces so that all incoming packets are processed by the load balancing logic. Moreover, it pushes the

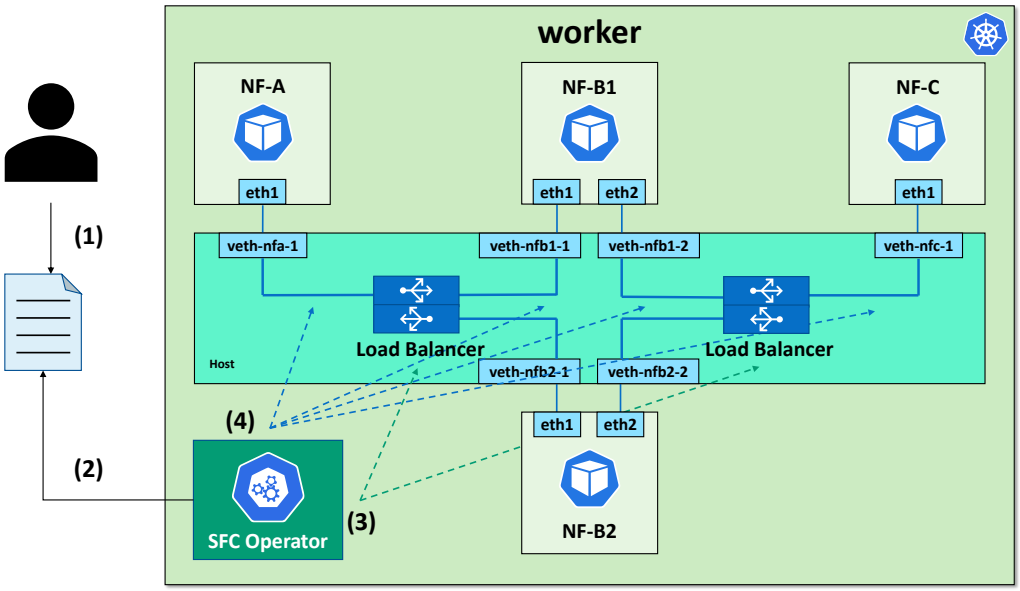


Fig. 6.3 Chain creation process

interfaces indexes in the load balancer data structures so that they know to which NFs they are linked.

As concerns the deletion, when a user deletes the `ServiceFunctionChain` resource from the cluster, the SFC operator reacts removing the eBPF Load Balancers injected in the kernel and cleaning up all the TC hooks of `veth` interfaces attached to the NF pods.

### 6.4.5 NF scaling

In order to properly handle scaling and more generally the creation and the deletion of NF pods, the system reacts to two specific events during the lifecycle of the pod. The first one is the beginning of pod execution and the second one is the start of the pod deletion procedure. When the SFC Operator notices that a new NF pod has entered the running phase, it immediately proceeds with the operations needed to include it in the chain. First, it searches for all the eBPF Load Balancers it must be linked to, then it proceeds with the actual connection of the NF pod interfaces as described previously. On the other hand, as soon as the SFC Operator notices that a NF pod is going to be deleted, it removes from load balancers all the information related to that replica so that it will not be selected by the load balancing logic

anymore. It is important to highlight that the above clean-up operations must occur before actually shutting down the pod in order to avoid dropping packets. This can be achieved by postponing the actual NF pod termination so that the SFC Operator has enough time to complete the cleanup, which can be done by leveraging the `PreStop` hook in the containers lifecycle, that allows the execution of a command before starting the termination phase.

## 6.5 Evaluation

This section presents a preliminary evaluation of the proposed architecture in terms of performance and reaction times. First, we compare our SFC scaling approach that operates on individual NFs against the alternative approach based on the scaling of the whole chain. Second, we evaluate the performance of the eBPF Load Balancers in terms of their throughput. Third, we measure the reaction time of the overall system at the occurrence of scaling events.

### 6.5.1 NF scaling efficiency

In our solution, the scaling of NFs is based on the autoscaling features of Kubernetes, whose logic operates on each single NF deployment. This enables SFCs that can scale only the component that is stressed or underused, allowing better resource usage compared to other approaches that scale the whole chain. To highlight this aspect, we measured the amount of resources requested by an SFC when it scales out only a stressed NF as opposed to when it scales out the entire chain. The tested SFC was composed of three NFs: a firewall, a simple pass-through traffic policer, and a gateway. In terms of computational resources they requested[2] 200, 150, 100 milliCPU respectively for each instance. The firewall is the NF that is assumed to scale.

Figure 6.4 shows the results of the test in terms of milliCPU *requested* by the entire chain for the two approaches as the number of replicated instances increase. In case of interrupt-based NFs, the *requested* CPU does not imply it is actually consumed, hence it could be re-assigned to other services demanding more CPU

[2]In Kubernetes, *requests* determine the amount of resources reserved for a pod, which may be different from the consumed resource, which is usually smaller than the value above.

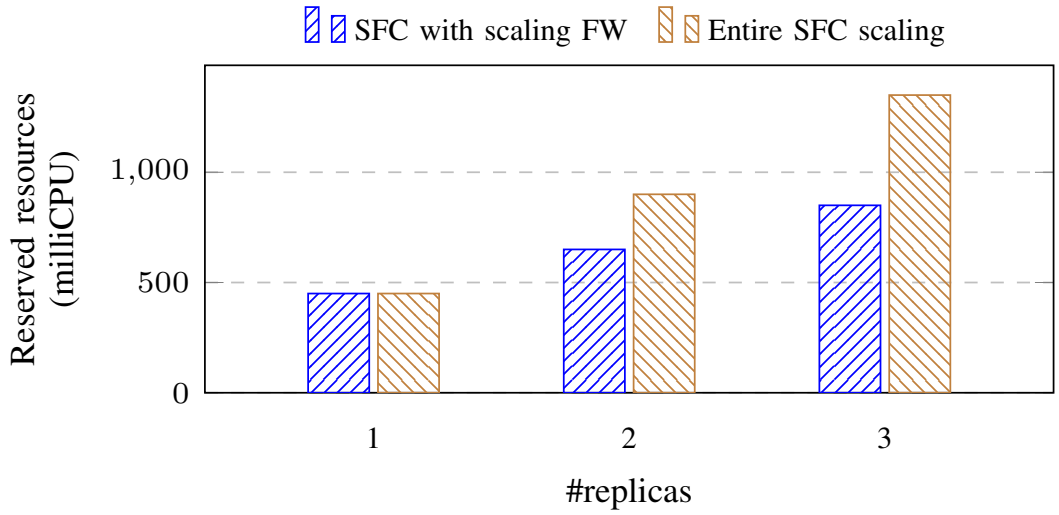


Fig. 6.4 Requested milliCPU as the number of scaled instance increase.

power. However, this is possible only in case the over-allocation policy is allowed, which is usually discouraged in case of strong service guarantees. Vice versa, with NFs based on a busy-polling model (e.g., in case of DPDK-based NFs), the amount of CPU actually consumed is equal to the reserved value, independently of the amount of traffic to be processed, hence leading to an unnecessary waste of resources. Hence, this result confirms that the scaling performed on individual NFs allows to greatly decrease the resource utilization, avoiding unnecessary over-allocations or resources wasted by busy-polling NFs.

### 6.5.2 eBPF Load Balancer performance

In the traditional approach in which the whole chain is scaled, NFs can be directly connected through a point-to-point link (e.g., `veth`). Instead, in our architecture the fan-in/fan-out of each NF can be potentially greater than one, hence requiring the presence of a load balancer between NFs, which may introduce additional performance penalties. This section evaluates this additional cost of the eBPF load balancer, compared to other two interconnection mechanisms provided by Linux kernel: Virtual Ethernet (`veth`) and the standard *Linux bridge* software. In this test, physical connections between NF pods are created as follows:

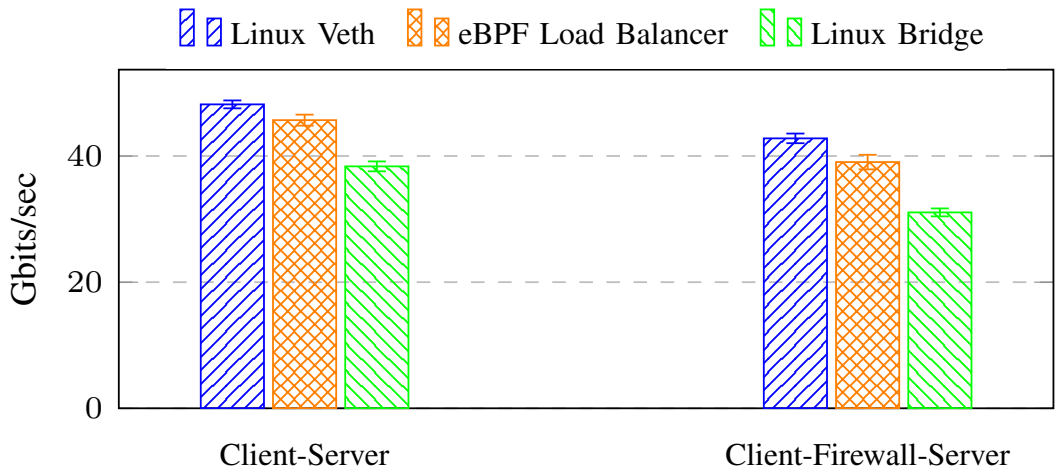


Fig. 6.5 Performance comparison between scenarios' sub-cases.

- Virtual ethernet: the `veth` is used as a direct link between the NFs. One end of the `veth` is placed in the network namespace of the first NF and the other end is placed in the network namespace of the second NF.
- eBPF Load Balancer: each NF is connected to the host network namespace through a `veth` and the load balancer cross-connects the `veth` ends on the host side.
- Linux bridge: similar to the previous case, but in this case the `veth` head ending in the host network namespace are connected through a Linux bridge.

Tests address two different scenarios: (*i*) a chain composed of two pods, a client and a server; (*ii*) a chain composed of three pods, a client, a transparent firewall and a server. In these tests, each NF is deployed with a single replica. In both scenarios all the pods of the chain are executed on the same Kubernetes node, with the physical interconnections implemented through the above technologies. The throughput of each chain has been measured with `iperf3`, using TCP traffic with standard parameters.

Results in Figure 6.5 show that the additional overhead introduced by the eBPF load balancer is very limited compared to the optimal case (`veth`), and much better than any alternative technologies, such as Linux bridge. This is due to the efficiency provided by the eBPF technology, which operates directly in-kernel, thus avoiding expensive context switching. Hence, despite the proof-of-concept (session-based) load-balancing logic, our implementation proved to be definitely faster than the standard Linux bridge.

### 6.5.3 Reaction Time

In order to evaluate the reaction time of the system, we measured the time required from a change of state of a replica to be noticed by the operator, and also how long it takes to the operator to configure/clean-up the interfaces and update the load balancing rules affected by the event. For the tests, a generic NF with two network interfaces has been used. For this reason, the add or remove operations on the interfaces are performed twice for each event. The time taken to update the load balancers eBPF maps is not shown in the results because its contribution is so low that can be considered negligible.

The first test evaluates the time elapsed from starting a pod (representing a new replica) until it is fully included in the chain by the operator. Results in Figure 6.6 show that most of the delay is due to the propagation of the event related to the status of the running pod in the cluster, which falls under the responsibility of the standard Kubernetes logic, while the time required by our operator is much smaller. More in depth, the graph shows that, within the operator, the most time-consuming operations are the configuration of the two TC hook filters (before/after the NF) that connect the current NF to the rest of the chain through the eBPF load balancing logic.

The reverse case is the scale-in operation, which corresponds to the removal of one replica. In this case, the operator reacts as soon as it detects that a pod is going to be deleted. In this test we measured the time taken by the SFC to converge to the new state, starting from the instant in which the pod begins its graceful termination to when it is successfully removed from the chain and all the cross-connections are properly updated. The pod used in the tests was forced to delay the beginning of the termination procedure by 0.5 seconds.

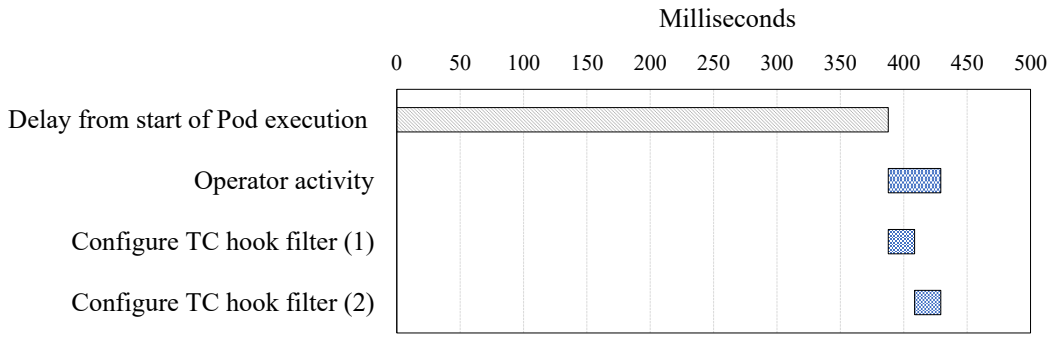


Fig. 6.6 Reaction time composition for a new replica event.

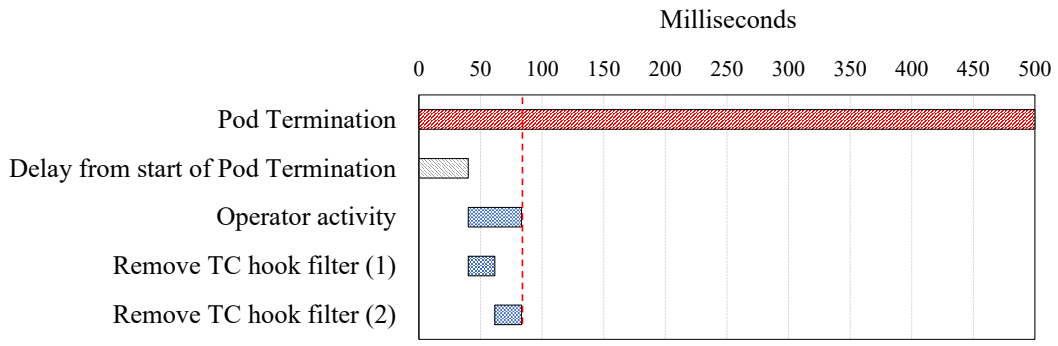


Fig. 6.7 Reaction time composition for a terminating replica event.

Figure 6.7 shows that the time spent by the operator is similar to the previous test. In this case the main contribution is given by the removal of TC hook filter from the `veth` interfaces. Even in this case the operator reacts after a slight delay as evidenced by the grey bar in the picture, due also in this case to the time to propagate the deletion event in the cluster. The dashed line highlights the moment in which the replica is no longer part of the chain, showing that the SFC converges much faster than the time required by Kubernetes to terminate the pod. This is a desired result, as the operator must complete his operations before the pod terminates in order to prevent traffic from being forwarded to a replica that no longer exists.

Finally, additional tests (not shown here for the sake of brevity) confirm that no packets are lost in the reconvergence process, hence making this operation loss-free. However, some traffic sessions are handled by a different replica than before, with possible problems in case of stateful NFs. A clever mechanism that avoids session redirections, which requires a smarter load balancer and a more sophisticated pod termination logic, is left to our future work.

## 6.6 Conclusions

This chapter presented a possible approach for highly scalable SFCs in a cloud-native environment such as Kubernetes. In particular, we proposed a model to represent the concept of scalable SFCs and a PoC implementation to integrate them into the aforementioned platform. This work demonstrates the possibility to support SFCs with an arbitrary number of NF replicas that can change continuously over time, thus

enabling the creation of chains that can dynamically adapt to the traffic load, with support for different L2/L3 network models.

Our experimental evaluation has shown that our solution achieves better results in terms of overall resource consumption compared to other approaches based on the scaling of whole SFCs. The performance evaluation of our eBPF-based cross-connections have shown that the maximum throughput is comparable with simpler direct links, emphasizing the low overhead deriving from the additional load balancing logic.

As part of our future work, we foresee the possibility to extend the load balancing implementation with a more advanced weighted logic based on the effective load on NFs, which could further optimize the use of NFs resources and improve the average quality of services offered by SFCs.

# Part V

# Concluding Remarks

# Chapter 7

# Concluding Remarks

In this dissertation, we explored the challenges faced by edge computing deployments in supporting the increasing volume of network traffic and cloud applications that seek the latency, throughput, and security advantages promised by moving computation to the network edge. We addressed the problem of resource optimization from different directions, in order to provide optimal support for the heterogeneous kinds of traffic and types of applications that rely on edge data centers. We first made the case for sharing the same physical nodes to run both cloud-native applications and telco provider network functions, underlining how a rigid partitioning of servers between the two kinds of workloads is not feasible due to the limited amount of resources available. In this respect, by designing a proof-of-concept 5G UPF we showed how running eBPF-based telco-grade network functions in-kernel is possible and allows a higher degree of integration and easier resource sharing with cloud-native applications relying on the kernel TCP/IP stack, unlike other widespread kernel-bypass frameworks. We then explored the capabilities of the XDP/AF_XDP systems of the Linux kernel to flexibly steer traffic in user space or process it in kernel space, showing how a hybrid approach can be used to maximize the throughput according to the kind of traffic being processed. To allow multiple customers to leverage the advantages of edge computing in an efficient way we addressed the problem of multi-tenancy and designed SURE, a fast and secure serverless framework for microservices. Leveraging a light and secure virtualization environment based on Unikernels, and enhancing it with a fast zero-copy data plane based on shared memory and a low-overhead library-based sidecar, both hardened with hardware-based memory protection mechanisms, SURE guarantees the isola-

tions required by customers when running their workloads in public environments, without compromising on performance. At last, we addressed the problem of service orchestration with a focus on their interconnection. We showed how the use of eBPF to interconnect application modules can be simplified without reduction of performance, by designing a Kubernetes network provider composed of reusable eBPF modules coming from the traditional networking world, and we extended the automatic scaling capabilities of Kubernete to network function chains, by designing a chaining mechanism based on scalable cross-connections.

While this dissertation mainly focused on different aspects of resource optimization in isolation we plan, as a future work, to design a holistic system for resource orchestration at the edge that, by combining the insight and innovations presented in this work, can orchestrate the heterogeneous set of application and patterns of traffic present at the edge achieving, at the same time, efficiency and optimaility with respect to the selected objective function.

# References


[1] https://istio.io/, 2024. [ONLINE].

[2] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting service mesh overheads, 2022.

[3] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 780–794, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Thomas Graf. How eBPF will solve Service Mesh – Goodbye Sidecars. https://isovalent.com/blog/post/2021-12-08-ebpf-servicemesh/, 2024. [ONLINE].

[5] Mansoor Shafi, Andreas F. Molisch, Peter J. Smith, Thomas Haustein, Peiying Zhu, Prasan De Silva, Fredrik Tufvesson, Anass Benjebbour, and Gerhard Wunder. 5g: A tutorial overview of standards, trials, challenges, deployment, and practice. *IEEE Journal on Selected Areas in Communications*, 35(6):1201–1221, 2017.

[6] Lalit Chettri and Rabindranath Bera. A comprehensive survey on internet of things (iot) toward 5g wireless systems. *IEEE Internet of Things Journal*, 7(1):16–32, 2020.

[7] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. *IEEE Communications Surveys & Tutorials*, 19(3):1657–1681, 2017.

[8] Pasika Ranaweera, Anca Delia Jurcut, and Madhusanka Liyanage. Survey on multi-access edge computing security and privacy. *IEEE Communications Surveys & Tutorials*, 23(2):1078–1124, 2021.

[9] Luigi Rizzo. netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[10] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Packetmill: toward per-core 100-gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Dpdk. https://www.dpdk.org/, 2024. [ONLINE].

[12] Pf_ring zc (zero copy). https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/. Accessed: 2024-02-21.

[13] Danielle E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, 2016.

[14] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the v out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 203–216, Savannah, GA, November 2016. USENIX Association.

[15] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, page 26–31, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16, 2015.

[17] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

[18] F-Stack. https://www.f-stack.org/, 2024. [ONLINE].

[19] Matt Fleming. A thorough introduction to ebpf, 2017.

[20] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David 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*, CoNEXT '18, pages 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Bernal, Yunsong Lu, and Jianwen Pi. Securing linux with a faster and scalable iptables. *SIGCOMM Comput. Commun. Rev.*, 2019.

[22] Gilberto Bertin. Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2, pages 1–5. The NetDev Society, 2017.

[23] Katran. https://github.com/facebookincubator/katran. (Accessed on: Feb. 8, 2022).

[24] Daniel J Walsh. Are docker containers really secure? https://opensource.com/business/14/7/docker-security-selinux, 2014. [ONLINE].

[25] Federico Parola, Fulvio Risso, and Sebastiano Miano. Providing telco-oriented network services with ebpf: the case for a 5g mobile gateway. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 221–225, 2021.

[26] Federico Parola, Roberto Procopio, Roberto Querio, and Fulvio Risso. Comparing user space and in-kernel packet processing for edge data centers. *SIGCOMM Comput. Commun. Rev.*, 53(1):14–29, apr 2023.

[27] Federico Parola, Leonardo Di Giovanna, Giuseppe Ognibene, and Fulvio Risso. Creating disaggregated network services with ebpf: the kubernetes network provider use case. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pages 254–258, 2022.

[28] Francesco Monaco, Giuseppe Ognibene, Federico Parola, and Fulvio Risso. Enabling scalable sfcs in kubernetes with ebpf-based cross-connections. In *2022 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 33–38, 2022.

[29] Tamás Lévai, Gergely Pongrácz, Péter Megyesi, Péter Vörös, Sándor Laki, Felicián Németh, and Gábor Rétvári. The price for programmability in the software data plane: The vendor perspective. *IEEE Journal on Selected Areas in Communications*, 36(12):2621–2630, 2018.

[30] Suneet Kumar Singh, Christian Esteve Rothenberg, Gyanesh Patra, and Gergely 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*, pages 9–14, 2019.

[31] DongJin Lee, JongHan Park, Chetan Hiremath, John Mangan, and Michael Lynch. Towards achieving high performance in 5g mobile packet core's user plane function. Technical report, Intel Corporation, SK Telecom, 2018.

[32] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A framework for ebpf-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management*, 18(1):133–151, 2021.

[33] Quentin Monnet. Stateful packet processing: two-color token-bucket poc in bpf, 2017.

[34] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[35] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[36] TIPSY Authors. Tipsy: Telco pipeline benchmarking system, 2018.

[37] Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong. Accelerating virtual network functions with fast-slow path architecture using express data path. *IEEE Transactions on Network and Service Management*, 17(3):1474–1486, 2020.

[38] Thiago A. Navarro do Amaral, Raphael V. Rosa, David F. Cruz Moura, and Christian E. Rothenberg. An in-kernel solution based on xdp for 5g upf: Design, prototype and performance evaluation. In *2021 17th International Conference on Network and Service Management (CNSM)*, pages 146–152, New York, NY, USA, 2021. IEEE.

[39] Federico Parola, Fulvio Risso, and Sebastiano Miano. Providing telco-oriented network services with ebpf: the case for a 5g mobile gateway. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 221–225, New York, NY, USA, 2021. IEEE.

[40] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8, New York, NY, USA, 2018. IEEE, IEEE.

[41] Jonathan Corbet. Af_xdp, 2018.

[42] Björn Töpel. Af_xdp, zero-copy support, 2018.

[43] Björn Töpel. Introduce preferred busy-polling, 2020.

[44] Linux kernel documentation. Scaling in the linux networking stack, 2022.

[45] Jonathan Corbet. Receive packet steering, 2009.

[46] Jake Edge. Receive flow steering, 2010.

[47] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 337–350, New York, NY, USA, 2012. Association for Computing Machinery.

[48] Clayne B Robison. How to set up intel® ethernet flow director, 2017.

[49] Magnus Karlsson and Björn Töpel. The path to dpdk speeds for af xdp. In *Linux Plumbers Conference*, San Francisco, California, 2018. Linux foundation.

[50] Magnus Karlsson. add need_wakeup flag to the af_xdp rings, 2019.

[51] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 673–689, Berkeley, CA, USA, 2020. USENIX Association.

[52] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, aug 2004.

[53] Peng Zheng, Wendi Feng, Arvind Narayanan, and Zhi-Li Zhang. Nfv performance profiling on multi-core servers. In *2020 IFIP Networking Conference (Networking)*, pages 91–99, New York, NY, USA, 2020. IEEE, IEEE.

[54] Paul McKenney. What is rcu, fundamentally?, 2007.

[55] William Tu. Af_xdp support for veth, 2018.

[56] Jeffrey T Kirsher. i40e/i40evf: Use build_skb to build frames, 2017.

[57] Jesper Dangaard Brouer. bpf-examples - af_xdp-interaction, 2022.

[58] Oliver Hohlfeld, Johannes Krude, Jens Helge Reelfs, Jan Rüth, and Klaus Wehrle. Demystifying the performance of xdp bpf. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 208–212, New York, NY, USA, 2019. IEEE.

[59] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the open vswitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 245–257, New York, NY, USA, 2021. Association for Computing Machinery.

[60] Marcelo Abranches and Eric Keller. A userspace transport stack doesn't have to mean losing linux processing. In *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 84–90, New York, NY, USA, 2020. IEEE, IEEE.

[61] Farbod Shahinfar, Sebastiano Miano, Alireza Sanaee, Giuseppe Siracusano, Roberto Bifulco, and Gianni Antichi. The case for network functions decomposition. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, pages 475–476, New York, NY, USA, 2021. Association for Computing Machinery.

[62] Jesper Dangaard Brouer. Xdp-hints: Xdp gaining access to hw offload hints via btf, 2022.

[63] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 412–426, New York, NY, USA, 2021. Association for Computing Machinery.

[64] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, Boston, MA, July 2023. USENIX Association.

[65] What is FaaS? https://www.ibm.com/topics/faas, 2024. [ONLINE].

[66] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[67] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. Executing microservice applications on serverless, correctly. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.

[68] Implementing Microservices on AWS. https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices-on-aws.html, 2024. [ONLINE].

[69] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 2019.

[70] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. Xfaas: Hyperscale and low cost serverless functions at meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 231–246, New York, NY, USA, 2023. Association for Computing Machinery.

[71] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. Ditto: Efficient serverless analytics with elastic parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 406–419, New York, NY, USA, 2023. Association for Computing Machinery.

[72] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote procedure call as a managed system service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 141–159, Boston, MA, April 2023. USENIX Association.

[73] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1489–1504, Boston, MA, April 2023. USENIX Association.

[74] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gVisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[75] gvisor: The Container Security Platform. https://gvisor.dev/, 2024. [ONLINE].

[76] Alessandro Randazzo and Ilenia Tinnirello. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 209–214, 2019.

[77] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[78] Bo Tan, Haikun Liu, Jia Rao, Xiaofei Liao, Hai Jin, and Yu Zhang. Towards lightweight serverless computing via unikernel as a function. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2020.

[79] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[80] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. *SIGPLAN Not.*, 39(1):291–304, mar 2011.

[81] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery.

[82] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—Optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association.

[83] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework? *Queue*, 11(11):30–44, dec 2013.

[84] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[85] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[86] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. KylinX: A dynamic library operating system for simplified and efficient cloud virtualization. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 173–186, Boston, MA, July 2018. USENIX Association.

[87] Compatibility of Unikraft. https://unikraft.org/docs/concepts/compatibility, 2024. [ONLINE].

[88] Kubernetes. https://kubernetes.io/, 2024. [ONLINE].

[89] runu OCI runtime. https://unikraft.org/docs/getting-started/integrations/container-runtimes, 2024. [ONLINE].

[90] NanoVMs. https://nanovms.com/, 2024. [ONLINE].

[91] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, mar 2017.

[92] Memory protection keys. https://lwn.net/Articles/643797/, 2024. [ONLINE].

[93] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 164–177, New York, NY, USA, 2003. Association for Computing Machinery.

[94] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. California, USA, 2005.

[95] Jesse Hertz. Abusing privileged and unprivileged linux containers. *Whitepaper, NCC Group*, 48, 2016.

[96] Aaron Grattafiori. Understanding and hardening linux containers. *Whitepaper, NCC Group*, 2016.

[97] Anjali, Tyler Caraza-Harter, and Michael M. Swift. Blending containers and virtual machines: A study of firecracker and gvisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 101–113, New York, NY, USA, 2020. Association for Computing Machinery.

[98] Shixiong Qi, Ziteng Zeng, Leslie 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*, pages 1–1, 2023.

[99] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[100] Lianjie Cao and Puneet Sharma. Co-locating containerized workload using service mesh telemetry. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, page 168–174, New York, NY, USA, 2021. Association for Computing Machinery.

[101] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[102] Drew Dean and Alan J Hu. Fixing races for fun and profit: how to use access (2). In *USENIX security symposium*, pages 195–206, 2004.

[103] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E. Carlson, and Prateek Saxena. Elasticlave: An efficient memory model for enclaves. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4111–4128, Boston, MA, August 2022. USENIX Association.

[104] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. Leveraging service meshes as a new network layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, page 229–236, New York, NY, USA, 2021. Association for Computing Machinery.

[105] Michael Hofmann. Service mesh vs. framework: resilience in distributed systems with isio or hystrix. https://devm.io/microservices/resilience-isio-hystrix, 2024. [ONLINE].

[106] Service mesh and service discovery. https://www.nginx.com/learn/service-mesh/, 2024. [ONLINE].

[107] Mohammad Reza Saleh Sedghpour, Cristian Klein, and Johan Tordsson. An empirical study of service mesh traffic management policies for microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, page 17–27, New York, NY, USA, 2022. Association for Computing Machinery.

[108] Joshua Levin and Theophilus A. Benson. Viperprobe: Rethinking microservice observability with ebpf. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–8, 2020.

[109] Unix domain socket. https://en.wikipedia.org/wiki/Unix_domain_socket, 2024. [ONLINE].

[110] Cilium. https://cilium.io/, 2024. [ONLINE].

[111] Linux Programmer's Manual. tc-bpf(8). https://man7.org/linux/man-pages/man8/tc-bpf.8.html, 2024. [ONLINE].

[112] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, Boston, MA, April 2023. USENIX Association.

[113] Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach. Usetl: Unikernels for serverless extract transform and load why should you settle for less? In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, page 23–30, New York, NY, USA, 2019. Association for Computing Machinery.

[114] Stefan Lankes, Simon Pickartz, and Jens Breitbart. Hermitcore: A unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[115] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference*

*on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[116] Costin Lupu, Andrei Albiundefinedoru, Radu Nichita, Doru-Florin Blânzeanu, Mihai Pogonaru, Răzvan Deaconescu, and Costin Raiciu. Nephele: Extending virtualization environments for cloning unikernel-based vms. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 574–589, New York, NY, USA, 2023. Association for Computing Machinery.

[117] Guanyu Li, Dong Du, and Yubin Xia. Iso-unik: lightweight multi-process unikernel through memory protection keys. *Cybersecurity*, 3:1–14, 2020.

[118] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-In-Time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, Oakland, CA, May 2015. USENIX Association.

[119] Dan Williams and Ricardo Koller. Unikernel monitors: Extending minimalism outside of the box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[120] Giuseppe Siracusano, Roberto Bifulco, Simon Kuenzer, Stefano Salsano, Nicola Blefari Melazzi, and Felipe Huici. On the fly tcp acceleration with miniproxy. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, page 44–49, New York, NY, USA, 2016. Association for Computing Machinery.

[121] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: A library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 546–558, New York, NY, USA, 2021. Association for Computing Machinery.

[122] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. Flexos: Towards flexible os isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 467–482, New York, NY, USA, 2022. Association for Computing Machinery.

[123] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 445–458, Seattle, WA, April 2014. USENIX Association.

[124] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 90–103, New York, NY, USA, 2019. Association for Computing Machinery.

[125] Hao Li, Yihan Dang, Guangda Sun, Guyue Liu, Danfeng Shan, and Peng Zhang. LemonNFV: Consolidating heterogeneous network functions at line speed. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1451–1468, Boston, MA, April 2023. USENIX Association.

[126] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[127] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[128] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process isolation for High-Throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[129] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.

[130] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 143–156, New York, NY, USA, 2020. Association for Computing Machinery.

[131] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient In-Process isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[132] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 936–952, Boston, MA, August 2022. USENIX Association.

[133] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association.

[134] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129 vol.2, Jan 2000.

[135] Donald Ray and Jay Ligatti. Defining code-injection attacks. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 179–190, New York, NY, USA, 2012. Association for Computing Machinery.

[136] DPDK Poll Mode Driver. https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html, 2024. [ONLINE].

[137] Jeffrey C Mogul and Kadangode K Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[138] send(2) — Linux manual page. https://man7.org/linux/man-pages/man2/send.2.html, 2024. [ONLINE].

[139] recv(2) — Linux manual page. https://man7.org/linux/man-pages/man2/recv.2.html, 2024. [ONLINE].

[140] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[141] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 693–707, New York, NY, USA, 2018. Association for Computing Machinery.

[142] Write XOR eXecute. https://en.wikipedia.org/w/index.php?title=W%5EX&oldid=1145060869, 2024. [ONLINE].

[143] Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo, 2024. [ONLINE].

[144] lwIP - A Lightweight TCP/IP stack. https://savannah.nongnu.org/projects/lwip/, 2024. [ONLINE].

[145] NGINX Service Mesh. https://www.nginx.com/products/nginx-service-mesh/, 2024. [ONLINE].

[146] Locust: A Modern Load Testing Framework. https://locust.io/, 2024. [ONLINE].

[147] Knative. https://knative.dev/docs/, 2024. [ONLINE].

[148] Cni – the container network interface. https://github.com/containernetworking/cni. (Accessed on: Jan. 30, 2022).

[149] Flannel. https://github.com/flannel-io/flannel. (Accessed on: Apr. 12, 2022).

[150] Calico. https://github.com/projectcalico/calico. (Accessed on: Feb. 8, 2022).

[151] Telecom User Group. Cloud native thinking for telecommunications, 2020.

[152] Karamjeet Kaur, Veenu Mangat, and Krishan Kumar. A comprehensive survey of service function chain provisioning approaches in sdn and nfv architecture. *Computer Science Review*, 38:100298, 2020.

[153] Boutheina Dab, Ilhem Fajjari, Mathieu Rohon, Cyril Auboin, and Arnaud Diquélou. Cloud-native service function chaining for 5g based on network service mesh. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pages 1–7, 2020.

[154] Adel Bouridah, Ilhem Fajjari, Nadjib Aitsaadi, and Hacene Belhadef. Optimized scalable sfc traffic steering scheme for cloud native based applications. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6, 2021.

[155] The Network Service Mesh Authors. Network service mesh.

[156] The Contiv-VPP Authors. Contiv-vpp.

[157] The Multus Authors. Multus.

[158] Kopf Authors. Kubernetes operator pythonic framework (kopf).

[159] The BCC Authors. Bpf compiler collection (bcc).

# Appendix A

# List of Publications

- **Federico Parola**, Roberto Procopio, Roberto Querio, and Fulvio Risso. "Comparing User Space and In-Kernel Packet Processing for Edge Data Centers." *ACM SIGCOMM Computer Communication Review 53*, no. 1 (2023): 14-29.
- **Federico Parola**, Leonardo Di Giovanna, Giuseppe Ognibene, and Fulvio Risso. "Creating Disaggregated Network Services with eBPF: the Kubernetes Network Provider Use Case." In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pp. 254-258. IEEE, 2022.
- Francesco Monaco, Giuseppe Ognibene, **Federico Parola**, and Fulvio Risso. "Enabling Scalable SFCs in Kubernetes with eBPF-based Cross-Connections." In *2022 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pp. 33-38. IEEE, 2022.
- **Federico Parola**, Roberto Procopio, and Fulvio Risso. "Assessing the performance of XDP and AF_XDP based NFs in edge data center scenarios." In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, pp. 481-482. 2021.
- **Federico Parola**, Fulvio Risso, and Sebastiano Miano. "Providing telco-oriented network services with eBPF: the case for a 5G mobile gateway." In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pp. 221-225. IEEE, 2021.
- **Federico Parola**, Sebastiano Miano, and Fulvio Risso. "A proof-of-concept 5g mobile gateway with ebpf." In *Proceedings of the SIGCOMM'20 Poster and Demo Sessions*, pp. 68-69. 2020.