Summer 2024

ENHANCING THE EFFICIENCY AND SCALABILITY OF CLOUD NETWORKING SYSTEMS

JIAXIN LEI
University of Texas at Arlington

ENHANCING THE EFFICIENCY AND SCALABILITY OF CLOUD NETWORKING SYSTEMS

by

JIAXIN LEI

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
August 2024

To my family.

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to my advisor, Dr. Hui Lu. He is a patient and supportive mentor, guiding students from broad research ideas to detailed designs. He is also a generous and kind group leader, providing us with state-of-the-art research facilities and a comfortable working environment. And he is an enthusiastic researcher, staying at the forefront of our research field with passion and curiosity. The experiences of working with Dr. Hui Lu have trained me to be a person with independent thought and professional attitude, which will influence my entire career.

I extend my sincere thanks to Dr. Jia Rao, who has provided guidance and collaboration since the beginning of my Ph.D. journey. His advocacy for high-quality research taste and insights into the academic career have greatly inspired me. I am also honored to have Dr. Song Jiang and Dr. Jiayi Meng serve on my committee. Their suggestions and feedback have been invaluable and deeply appreciated.

I am grateful for the collaboration with Kun Suo and Manish Munikar. The time spent working together has been both rewarding and memorable. Additionally, I am happy to have met many wonderful friends during my Ph.D.: Zhou Wang, Siming Huang, Shaofei Zhao, Baozhen Wang, Xinhai Zhang, Shengfu Zhang, Xia Cheng and Xiaoyu Zhang. I wish each of you success and happiness in your future endeavors.

I would like to express my special thanks to my parents, Xihong Yan and Hongmin Lei. They have always strived to provide me with the best possible education and have guided me in making pivotal decisions with their foresight. During the COVID-19 pandemic, they traveled to America twice to take care of my kids, giving

everything they could to help me. Without their support, I would not be where I am today. I also want to thank my parents-in-law, Hongru Liu and Jianhua Ren, for their constant support and care. Most importantly, I owe my every success to my wife and best friend, Zepin Ren. She has sacrificed a lot to take care of our family. She is always by my side with endless trust and unconditional love, no matter what kind of challenges and difficulties we face. Lastly, I thank my kids, Aaron Bolin Lei and Ava Chanyu Lei, who bring so much joy and happiness to our family.

July 23, 2024

ABSTRACT

ENHANCING THE EFFICIENCY AND SCALABILITY OF CLOUD NETWORKING SYSTEMS

JIAXIN LEI, Ph.D.

The University of Texas at Arlington, 2024

Supervising Professor: Hui Lu

Overlay networks are the de facto network virtualization technique for providing flexible and customized connectivity among distributed containers in the cloud. Despite their widespread adoption, overlay networks incur significant overhead due to their complexity, resulting in notable performance degradation compared to physical networks.

In this dissertation, I present our three-stage solutions aimed at addressing the challenges of efficiency and scalability in cloud-based container overlay networks: Firstly, we conduct a comprehensive empirical performance study of container overlay networks, identifying crucial parallelization bottlenecks within the kernel network stack. Our observations and root cause analysis uncover that these inefficiencies primarily arise from the increased complexity and prolonged packet processing paths introduced by additional network devices. Secondly, we propose Falcon, a fast and balanced container networking approach designed to scale the packet processing pipeline in overlay networks. Falcon pipelines software interrupts associated with different network devices of a single flow across multiple cores, thereby preventing the exe-

cution serialization of excessive software interrupts from overloading a single core. Additionally, Falcon supports multiple network flows by effectively multiplexing and balancing software interrupts among available cores. Lastly, we introduce MFLOW, a novel packet steering approach to parallelize the in-kernel data path of network flows. MFLOW exploits fine-grained packet-level parallelism by splitting packets of the same flow into multiple micro-flows, allowing parallel processing on multiple cores. This approach devises new generic mechanisms for flow splitting while preserving in-order packet delivery with minimal overhead.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

The rapid expansion of cloud services has driven a fundamental transformation in datacenter infrastructures. Container-based virtualization, as a successor of traditional virtual machine-based methods, offers a lightweight, process-based approach that has been widely embraced by cloud data centers for its superior portability, scalability, and agility. Containers facilitate higher server consolidation density and lower operational costs, leading to widespread industry adoption. Their underlying overlay networks have become the de facto networking technique for providing customized, flexible connectivity among distributed container-based services. Various container overlay network solutions are built upon the tunneling approaches, enabling container traffic to traverse physical networks via encapsulating container packets with host headers. This allows containers within the same virtual network to communicate in an isolated address space with private IP addresses, while routing packets through tunnels using the hosts' public IP addresses. Overlay networks are not only flexible and extensible but also support various network policies, such as isolation, rate limiting, and quality of service.

However, despite their advantages, container overlay networks introduce significant overheads compared to native host networks. Studies have shown that overlay networks achieve significantly lower throughput and suffer higher packet processing latency. The prolonged network packet processing path in overlay networks, which involves multiple namespaces and kernel network stacks, is a primary cause of this performance degradation. High-speed physical network devices further exacerbate

these issues, as the kernel must process packets rapidly to match the network speed. The inefficiency of existing parallelization mechanisms in the kernel network stack and the poor scalability of overlay networks on multi-core systems add to the challenge.

This dissertation aims to improve the efficiency and scalability of container overlay networks in cloud environments. Specifically, my research critically revisits established cloud infrastructures to identify overlooked system inefficiencies and proposes the redesign and implementation of systems consistent with contemporary cloud demands. In the following sections of this chapter, I will briefly introduce my contributions and outline the organization of this dissertation.

1.1 Contributions

To address the above challenges, this dissertation presents three key contributions focused on enhancing the efficiency and scalability of cloud-based container overlay networks.

1.1.1 Characterizing the Complex Behaviors of Container Overlay Networks

We conduct a comprehensive empirical performance study of container overlay networks [1] to uncover critical parallelization bottlenecks within the kernel network stack. Our detailed profiling and root cause analysis reveal that the high overhead and low efficiency of container overlay networks are complex and multifaceted:

First, high-performance, high-speed physical network devices (e.g., 40 and 100 Gbps Ethernet) require rapid packet processing by the kernel. However, the prolonged packet path in container overlay networks, involving multiple network devices, significantly slows down per-packet processing speed. More critically, we observe that modern operating systems only provide parallelization at the per-flow level rather

than per-packet. As a result, the maximum network throughput of a single container flow is limited by the processing capability of a single core (e.g., 6.4 Gbps for TCP in our case). Furthermore, while multi-core CPUs and multi-queue network interface cards (NICs) enable packets from different flows to be routed to separate CPU cores for parallel processing, container overlay networks exhibit poor scalability. Network throughput increases by only 4x despite a 6x increase in the number of flows. Additionally, under the same throughput conditions (e.g., 40 Gbps with 80 flows), overlay networks consume significantly more CPU resources (e.g., 2 to 3 times more). Our investigation identifies that this severe scalability issue is largely due to the inefficient interplay within the kernel among pipelined, asynchronous packet processing stages – an overlay packet traverses among the contexts of one hardware interrupt, three software interrupts, and the user-space process. With more flows, hardware inefficiencies such as poor cache efficiency and high memory bandwidth utilization become pronounced. Lastly, we observe that for flows with small packet sizes, these inefficiencies become even more severe in container overlay networks, which achieve only 50% of the packet processing rate of native hosts (e.g., for UDP packets). In addition to the prolonged network path, the high interrupt request (IRQ) rate and the associated high software interrupt (softirq) rate (i.e., 3x the IRQs) impair overall system efficiency by frequently interrupting running processes, resulting in enhanced context switch overhead.

1.1.2 **Accelerating Packet Processing via Device-Level Pipelining**

Overlay networks are essential in orchestrating communications among containers, yet they incur substantial performance losses when compared with host native networks. For instance, my research [1] indicated that a single container overlay flow experienced a 72% throughput degradation for TCP and a 58% degradation for UDP.

Such inefficiency primarily stems from the increased complexity introduced by overlay networks along the prolonged packet processing path, which includes additional network devices and consequently triggers numerous serialized software interrupts (e.g., container overlay networks incur 3x more software interrupts than host native networks). These interrupts can easily saturate a single CPU core. Unfortunately, existing in-kernel scaling methods are limited to distributing multiple flows across different CPU cores, thus proving inadequate for accelerating a single flow. Kernel's inefficiency and lack of parallelization mechanisms prevent these additional interrupts from being processed concurrently. Our in-depth investigations uncovered a previously unrecognized insight: packet processing within the context of a software interrupt is, in fact, associated with per individual network device. This discovery opened up new opportunities for scaling the execution of successive software interrupts from a single intensive container overlay flow across multiple CPU cores.

Motivated by this discovery, we developed FALCON [2], a fast and balanced container networking solution that parallelizes the processing of software interrupts associated with various network devices across multiple CPU cores. To achieve this, we implemented an innovative hashing mechanism in the kernel: It activates multiple transition functions on network devices, which perform re-hashing based on 5-tuple information, including IPs, ports, and network device IDs. This enables FALCON to allocate software interrupts from a single flow across various CPUs, assigning distinct hash values to different network devices for targeted CPU core processing. Additionally, FALCON maintains balance among multiple network flows (or processing pipelines) by monitoring system-wide CPU utilization and employing a two-choice, low-overhead algorithm for adaptive balancing (re-hashing only if the primary CPU choice is occupied) without constant CPU workload comparisons. The effectiveness of FALCON has been validated through comprehensive evaluations. Real-world ap-

plications demonstrated that FALCON significantly boosted throughput performance (increasing by up to 300% for web serving) and mitigated tail latency (reducing by up to 53% for data caching), while ensuring an even distribution of networking workloads throughout the system.

1.1.3 Accelerating Packet Processing via Packet-Level Parallelism

Equipped with state-of-the-art solutions like FALCON [2], the processing of a single container overlay flow can be split into multiple stages, each managed by a dedicated CPU core. Nonetheless, one limitation of FALCON is that its device-level pipelining approach is relatively coarse-grained. Despite the distribution of software interrupts over multiple CPU cores, a heavily utilized network device might still saturate a single CPU core, thus becoming a system bottleneck. A real-world scenario is: in TCP processing, intensive functions such as per-packet socket buffer (skb) allocation and generic receive offload (GRO) within the context of the first software interrupt can easily exceed one CPU core's processing capability. Although FALCON does suggest a function-level splitting to balance these tasks across cores, the skb allocation function alone can overload a single core. Hence, while advanced solutions like FALCON offer a certain degree of parallelization in packet processing, they fall short of addressing new bottlenecks.

To address the newly identified challenges, we proposed MFLOW [3], an innovative packet steering approach that exploits fine-grained, packet-level parallelism. MFLOW differentiates a single flow into multiple micro-flows, each processed independently on different CPU cores. Specifically, it routes skb allocation requests (treated as individual packets) within the NIC driver – immediately following hardware interrupts, to newly established per-core request ring buffers, subsequently initiating software interrupts on targeted CPU cores via inter-process interrupts (IPIs). Since

each micro-flow only contains a small batch of packets, the entirety of its processing stack, even though remaining on a single core, does not lead to a performance bottleneck. Another critical aspect of MFLOW is its flow reassembly mechanism. Given the diverse processing capacities of CPU cores, micro-flows may suffer from the potential sequence disorder, leading to reassembly overheads. MFLOW tackles this challenge with two strategies: First, micro-flows can be calibrated with an optimal batch size to substantially minimize sequence disruptions. Second, during dispatch, packets are tagged with batch numbers. The reassembly thread of MFLOW prioritizes a batch of packets for final-stage merging only when their sequence numbers align with the expected order, while out-of-order batches are held in per-core buffer queues. This batch-oriented reassembly method is markedly more efficient than Linux kernel's default per-packet reordering. Performance evaluations demonstrated that MFLOW significantly elevated the throughput of a single container overlay flow (e.g., by 81% in TCP and 139% in UDP) and amplified application-level performance (e.g., by up to 7.5x for web serving).

1.2 Dissertation Organization

This dissertation is organized into seven chapters, each detailing a different aspect of my research.

- In chapter 1, I introduce the background and motivation of this dissertation and provide an overview of my contributions.
- In chapter 2, I conduct a comprehensive empirical performance study of container overlay networks and root cause analysis within the kernel network stack to uncover critical parallelization bottlenecks.
- In chapter 3, I present Falcon, a fast and balanced container networking approach designed to scale the packet processing pipeline in overlay networks.

- In chapter 4, I introduce MFLOW, an innovative packet steering approach that exploits fine-grained, packet-level parallelism.
- In chapter 5, I review the related research, developments and technologies in the field of cloud networking systems.
- In chapter 6, I explore potential future research directions to address emerging challenges in cloud systems.
- In chapter 7, I summarize the key findings and contributions of this dissertation.

CHAPTER 2

TACKLING PARALLELIZATION CHALLENGES OF KERNEL NETWORK STACK FOR CONTAINER OVERLAY NETWORKS

Overlay networks are the de facto networking technique for providing flexible, customized connectivity among distributed containers in the cloud. However, overlay networks also incur non-trivial overhead due to its complexity, resulting in significant network performance degradation of containers. In this chapter, we perform a comprehensive empirical performance study of container overlay networks which identifies unrevealed, important parallelization bottlenecks of the kernel network stack that prevent container overlay networks from scaling. Our observations and root cause analysis cast light on optimizing the network stack of modern operating systems on multi-core systems to more efficiently support container overlay networks in light of high-speed network devices.

2.1 Introduction

As an alternative to virtual machine (VM) based virtualization, containers offer a lightweight process-based virtualization method for managing, deploying and executing cloud applications. Lightweight containers lead to higher server consolidation density and lower operational cost in cloud data centers, making them widely adopted by industry — Google even claims that "everything at Google runs in containers" [4]. Further, new cloud application architecture has been enabled by containers: services of a large-scale distributed application are packaged into separate containers, automatically and dynamically deployed across a cluster of physical or virtual ma-

chines with orchestration tools, such as Apache Mesos [5], Kubernetes [6], and Docker Swarm [7].

Container overlay networks are the de facto networking technique for providing customized connectivity among these distributed containers. Various container overlay network approaches are becoming available, such as Flannel [8], Weave [9], Calico [10] and Docker Overlay [11]. They are generally built upon the tunneling approach which enables container traffic to travel across physical networks via encapsulating container packets with their host headers (e.g., with the VxLAN protocol [12]). With this, containers belonging to a same virtual network can communicate in an isolated address space with their private IP addresses, while their packets are routed through "tunnels" using their hosts public IP addresses.

Constructing overlay networks in a container host can be simply achieved by stacking a pipeline of in-kernel network devices. For instance, for a VxLAN overlay, a virtual network device is created and assigned to a container's network namespace, while a tunneling VxLAN network device is created for packet encapsulation/decapsulation. These two network devices are further connected via a virtual switch (e.g., Open vSwitch [13]). Such a container overlay network is also extensible: various network policies (e.g., isolation, rate limiting, and quality of service) can be easily added to either the virtual switch or the virtual network device of a container.

Regardless of the above-mentioned advantages, container overlay networks incur additional, non-trivial overhead compared to the native host network (i.e., without overlays). Recent studies report that overlay networks achieve 50% less throughput than the native and suffer much higher packet processing latency [14, 15]. The prolonged network packet processing path in overlay networks can be easily identified as the main culprit. Indeed, as an example in the above VxLAN overlay network, a packet traverses three different namespaces (i.e., the container, overlay and host)

and two kernel network stacks (the container and host) in both sending and receiving ends, leading to high per-packet processing cost and long end-to-end latency. However, our investigation reveals that the causes of high-overhead and low-efficiency of container network overlays are much complicated and multifaceted:

First, the high-performance, high-speed physical network devices (e.g., 40 and 100 Gbps Ethernet) require the kernel to quickly process each packet (e.g., 300 ns for a 40 Gbps network link). However, as stated above, the prolonged packet path in container overlay networks slows down the per-packet processing speed with multiple network devices involved. More critically, we observe that modern OSes only provide parallelization of packet processing at the per-flow level (instead of per-packet); thus, the maximum network throughput of a single container flow is limited by the processing capability of a single core (e.g., 6.4 Gbps for TCP in our case).

Further, the combination of multi-core CPUs and multi-queue network interface cards (NIC) allows packets of different flows to route to separate CPU cores for parallel processing. Unfortunately, container overlay networks are observed to produce poor scalability — the network throughput increases by 4x with 6x number of flows. In addition, under the same throughput (e.g., 40 Gbps with 80 flows), overlay networks consume much more CPU resources (e.g., $2 \sim 3$ times). Our investigation finds that this severe scalability issue is largely due to the inefficient interplay by kernel among pipelined, asynchronous packet processing stages — an overlay packet traverses among the contexts of one hardware interrupt, three software interrupts and the user-space process. With more flows, the hardware also becomes inefficient with poor cache efficiency and high memory bandwidth.

Last, research has long observed inefficiency in the kernel network stack for flows with small packet sizes. We observe that such inefficiency becomes more severe in container overlay networks which achieve as low as 50% packet processing rate

of that in the native host (e.g., for UDP packets). We find that, in addition to prolonged network path processing path, the high interrupt request (IRQ) rate and the associated high software interrupt (softirq) rate (i.e., 3x of IRQs) impair the overall system efficiency by frequently interrupting running processes with enhanced context switch overhead.

In this chapter, we perform a comprehensive empirical performance study of container overlay networks and identify the above-stated new, critical parallelization bottlenecks in the kernel network stack. We further deconstruct these bottlenecks to locate their root causes. We believe our observations and root cause analysis will cast light on optimizing the kernel network stack to well support container network stacks on multi-core systems in light of high-speed network devices.

2.2 Background

In this section, we introduce the background of network packet processing (under Linux) and the existing optimizations for network packet processing.

2.2.1 Network Packet Processing

Packet processing traverses NICs, kernel space, and user space. Taking receiving a packet as an example (Figure 2.1): When a packet arrives at the NIC, it is copied (via DMA) to the kernel ring buffer and triggers a hardware interrupt (IRQ). The kernel responds to the interrupt and starts the receiving path. The receiving process in kernel is divided into two parts: the top half and the bottom half. The top half runs in the context of a hardware interrupt, which simply inserts the packet in the per-CPU packet queue and triggers the bottom half. The bottom half is executed in the form of a software interrupt (softirq), scheduled by the kernel at an appropriate time later and is the main routine that the packet is processed through the network

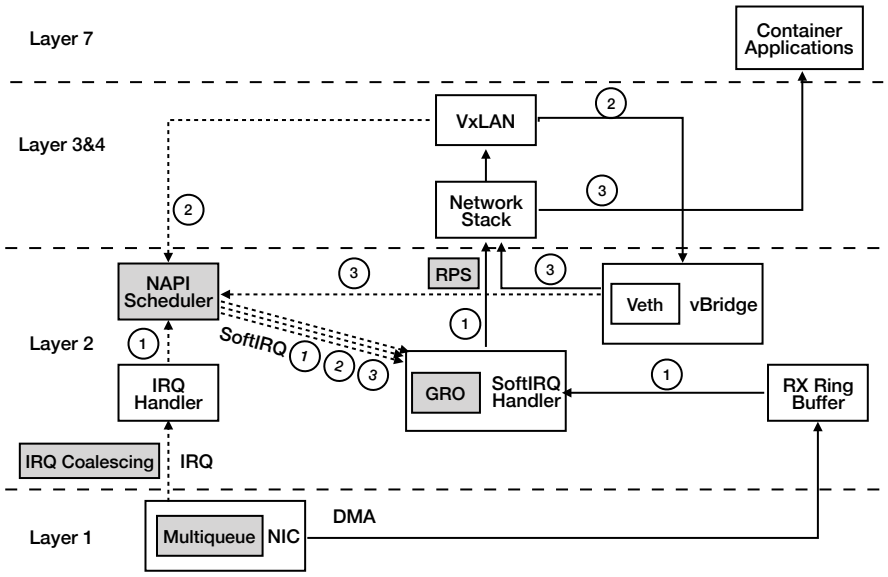

Figure 2.1: Illustration of data receiving path in Linux kernel.

protocol stack. After being processed at various protocol layers, the packet is finally copied to the user space buffer and passed to the application.

2.2.2 Container Overlay Networks

Overlay networks are a common way to virtualize container networks and provide customized connectivity among distributed containers. Container overlay networks are generally based on a tunneling technique (e.g., VxLAN): When sending a container packet, it encapsulates the packet in a new packet with the (source and destination) host headers; when receiving an encapsulated container packet, it decapsulates the received packet to recover the original packet and finally delivers it to the target container application by its private IP address.

As illustrated in Figure 2.1, the overlay network is created by adding additional devices, such as a VxLAN network device for packet encapsulation and decapsulation, virtual Ethernet ports (veth) for network interfaces of containers, and a virtual bridge

to connect all these devices. Intuitively, compared to the native host network, a container overlay network is more complex with longer data path. As an example in Figure 2.1, receiving one container packet raises one IRQ and *three* softirqs (by the host NIC, the VxLAN, the veth separately). In consequence, the container packet traverses three network namespaces (host, overlay and container) and two network stacks (container and host). Inevitably, it leads to high overhead of packet processing and low efficiency of container networking.

2.2.3 Optimizations for Packet Processing

There is a large body of work targeting at optimizing the kernel for efficient packet processing. We categorize them into two groups:

(1) Mitigating per-packet processing overhead: Packet processing cost generally consists of two parts: per-packet cost and per-byte cost. In modern OSes, per-packet cost dominates in packet processing. Thus, a bunch of optimizations have been proposed to mitigate per-packet processing including interrupt coalescing and polling-based approaches which reduce the number of interrupts [16, 17, 18]; packet coalescing which reduces the number of packets that need to be processed by kernel network stacks (e.g., Generic Receive Offload [19] and Large Receive Offload [20]); user-space network stacks which bypass the OS kernel thus reducing context switches [21]; and data path optimizations [22, 23, 24, 25].

(2) Parallelizing packet processing path: High-speed network devices can easily saturate one CPU core even with the above optimizations. This is especially true in virtualized overlay networks. To leverage multi-core systems, a set of hardware and software optimizations have been proposed to parallelize packet processing. Parallelism can be achieved using the hardware approach — a single physical NIC with multi-queues, each mapping IRQs to one separate CPU core with Receive Side Scal-

ing (RSS) [26]. Even without the NIC support, Receive Packet Steering (RPS) [27] can achieve the same RSS functionality in a software manner. Both RSS an RPS use hash functions (based on packet IP addresses and protocol ports) to determine the target CPU cores for packets of different flows. As we will show shortly, none of these approaches work effectively in container overlay networks.

2.3 Evaluation

In this section, we perform empirical studies to illustrate parallelization bottlenecks of the kernel network stack for container overlay networks.

2.3.1 Experimental Settings

We conducted experiments with three network configurations as follows:

- **Native Case.** Applications were running in the native host (i.e., no containers), and communicated with each other using the host IP addresses associated with the physical network interface — the traditional configuration in a non-virtualization, non-overlay environment.
- **Linux Overlay Case:** In this "transitional" case, we added one VxLAN software device attached to the host interface. Applications were still running in the native host, but communicated first through the VxLAN tunneling and then the host interface. We configured such a VxLAN device using the *iproute2* toolset [28].
- **Docker Overlay Case:** A Docker [29] overlay network was created to route container packets among hosts. Applications were running in Docker containers and communicated with each other using the containers' private IP addresses associated with the virtual interfaces (i.e., veth). A Linux bridge connected all local containers' veths and a VxLAN device (attached to the host interface).

The Docker overlay network requires a key-value database to store host network information and we chose *consul-0.5.2* [30] as the key-value store.

Notice that the packet processing path becomes longer from the native case to the docker overlay case.

Testbed Configurations. All experiments were conducted on two server machines each equipped with one Xeon E5-2630 v4 CPU (2.2 GHz and 10 physical cores with hyper-threading enabled — 20 virtual cores) and 64 GB memory. They were directly connected via a 40 Gb Mellanox ConnectX-3 Infiniband Network Interface Controller with the multi-queue technique enabled (16 packet queues). We ran *Docker-18.06* [29] on *Linux-16.04-4.4*, and used *iperf3* [31] as the benchmark applications. We have tuned the Linux network stack with all software optimizations enabled. To mimic a real setup, the MTU (maximum transfer unit) was set to 1,500 bytes by default. For all TCP and UDP experiments, the TCP packet size was set to 128 KB while the UDP packet size was set to 8 KB by default, unless otherwise stated. All experimental results were averaged over five or more runs.

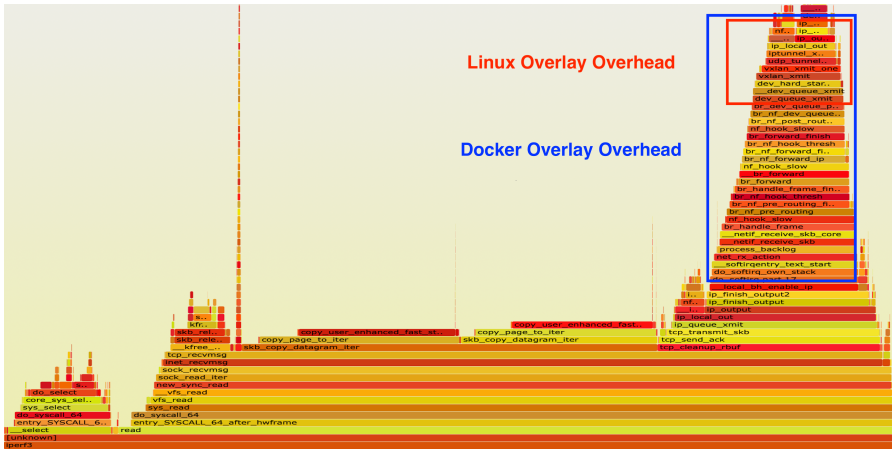

Figure 2.2: Function call graph along the TCP receiving path.

2.3.2 A Single Flow

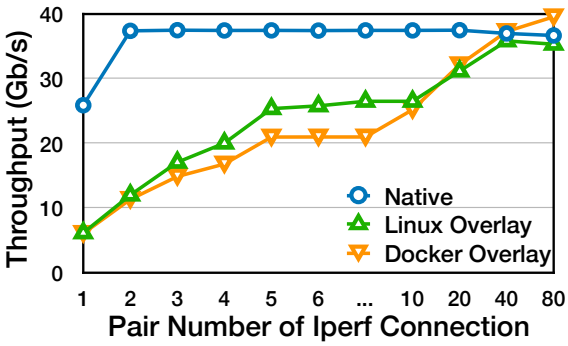

Figure 2.3: TCP throughput with varying pairs of connections.

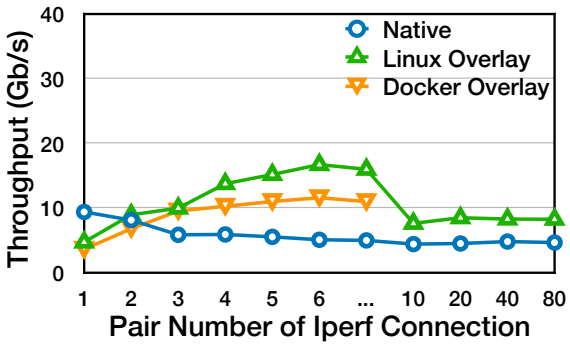

Figure 2.4: UDP throughput with varying pairs of connections.

First, we measure the TCP and UDP throughput using a single pair of iperf client and server residing on two machines separately.

Figure 2.3 shows the TCP throughput, while Figure 2.4 shows the UDP throughput. More specifically, the native case can reach around 23 Gbps for TCP and 9.3 Gbps for UDP. The Linux overlay performs a little better than the Docker overlay: in the Linux overlay, the TCP throughput reaches 6.5 Gbps, and the UDP reaches 4.7 Gbps. In comparison, in the Docker overlay case, the TCP throughput reaches around 6.4 Gbps, while the UDP throughput reaches only 3.9 Gbps. Compared to the native case, the throughput of the Docker overlay drops by 72% for TCP and 58% for UDP. As the packet processing path gets longer, the single pair bandwidth performance gets lower for both TCP and UDP cases.

The reason why the Docker overlay achieves much lower throughput than the native shows that: it consumes much higher CPU cycles for processing each packet. As plotted in Figure 2.5 (CPU usage breakdown for TCP) and Figure 2.6 (CPU usage breakdown for UDP), in the single flow case, the docker overlay consumes the same (or

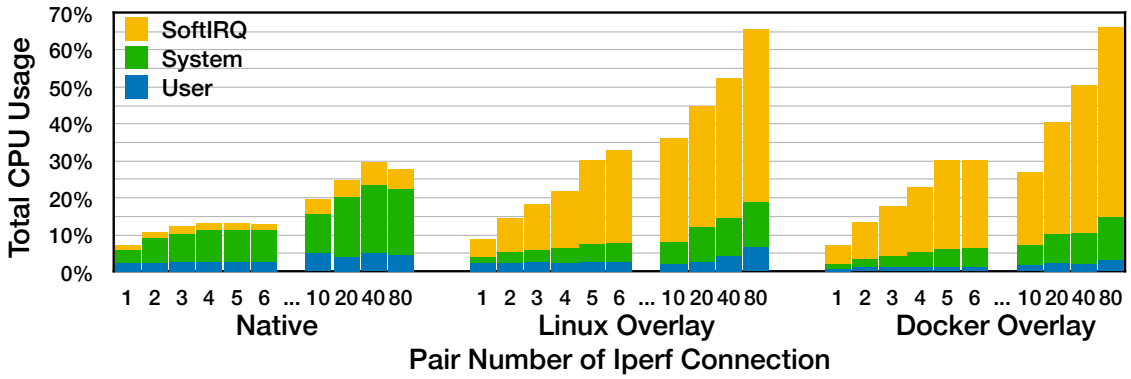

Figure 2.5: CPU usage breakdown on the receiver side with varying pairs of connections (TCP).

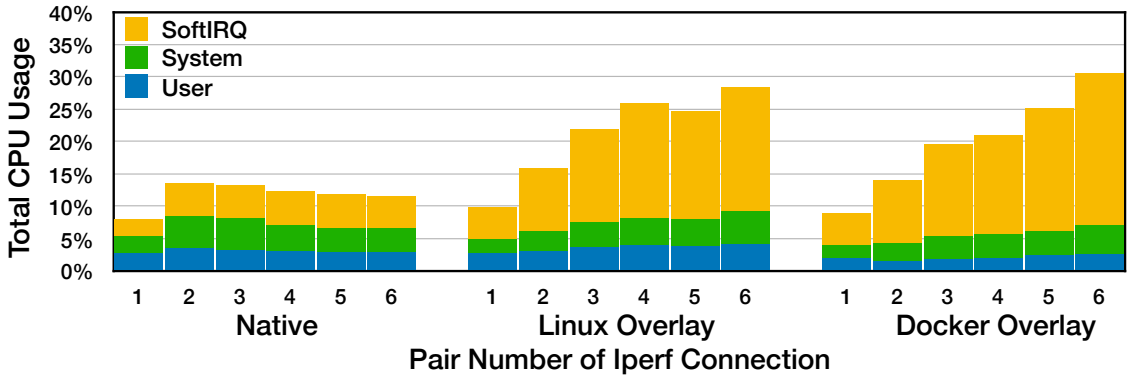

Figure 2.6: CPU usage breakdown on the receiver side with varying pairs of connections (UDP).

more) CPU usage with much less throughput, compared to the native case [1]. Figure 2.2 shows the function call stack along the TCP receiving path — the highlighted areas refer to the extra time spent in the functions of the overlay networks. It clearly demonstrates that the network processing path in the docker overlay network is much longer than the native case leading to extra CPU usage.

A question arises after we observe that the iperf client and server in the user space consume little CPU far away from occupying one single core: why cannot the

[1]Each machine has in total 20 virtual cores — 5% CPU usage means that a single core has been exhausted.

throughput keep scaling by consuming more CPU resources? Upon deeper investigation, we found that existing parallelization approaches (e.g., RSS or RPS) work at the per-flow level, as they decide the core for packet processing based on the flow-level information (i.e., IP addresses and/or port number). Hence, packets of the same flow are processed by the kernel on the same core — including all the softirqs triggered by all the network devices (i.e., the host interface, VxLAN and veth). As the docker overlay incurs longer packet processing path, it easily saturates one CPU core — as shown in Figure 2.5 and Figure 2.6, the CPU consumed by the kernel (i.e., the sum of the system and softirq parts) saturates one core.

2.3.3 **Multiple Flows**

As a single flow is far away from fully utilizing a 40 Gbps network link in the Docker overlay case, we tried to use multiple flows to saturate the network bandwidth by scaling the number of flows — we ran multiples pairs of iperf clients and servers from 1 to 80; each iperf client or server was running in a separate container.

As shown in Figure 2.3, we observe that the native case quickly reaches the peak throughput, ∼37 Gbps under TCP with only *two* pairs. However, the TCP throughput in the two overlay cases grows slowly as the pair number increases — the throughput increase by 4x (6.4 Gbps to 25 Gbps) with 6x number of pairs (1 pair to 6 pairs). Though all three cases can saturate the whole 40 Gbps network bandwidth (with 80 flows), under the same throughput (e.g., 40 Gbps) overlay networks consume much more CPU resources (e.g., around 2.5 times) than the native case.

This raises another question: why does the overlay network not scale well with multiple flows given that in this situation both RSS and RPS take effect (i.e., we did observe that packets of different flows were assigned with different CPU cores)? Our investigation shows that such a bad scalability is largely due to the inefficient

interplay of many packet processing tasks — IRQs, three different softirq contexts, and user-space processes. Too frequent context switches among these tasks greatly hurt the CPU cache efficiency, resulting in much higher memory bandwidth. For example, the Docker overlay case consumes 2x memory bandwidth with 50% network throughput with 7 pairs (not depicted in the figures). Such inefficiency can also be observed in Figure 2.5, though the total throughput does not scale, the CPU usage keeps increasing as the number of flow pairs increases — the kernel is just busy with juggling numerous tasks.

We observe the similar (and even worse) scalability in the UDP case [2] as illustrated in Figure 2.4 with the exception that the throughput of the native case keeps flat regardless of the flow numbers. The reason is that, in the native case, all UDP flows share the same flow-level information (i.e., same source and destination IP addresses); the RSS and RPS cannot distinguish them and assign all flows on the same core which is fully occupied. In contrast, in the overlay networks, the RSS and RPS can distinguish the packets of different flows by looking at the inner header information containing the private IP addresses of containers which are different among flows.

2.3.4 **Small Packets**

It is evident that most packets in the real world have small sizes (e.g., 80% $\leq$ 600 bytes [32]). The inefficient packet processing will negatively impact the performance of real-world applications. We conducted experiments to show the performance impact of overlay networks on small packets by varying the packet sizes of a single flow from 64 bytes to 8 KB. As illustrated in Figure 2.7, the Docker overlay performs a bit worse

[2]We cannot collect performance data after 7 pairs for the Docker overlay case, as the system becomes very unstable due to high packet drop rate.

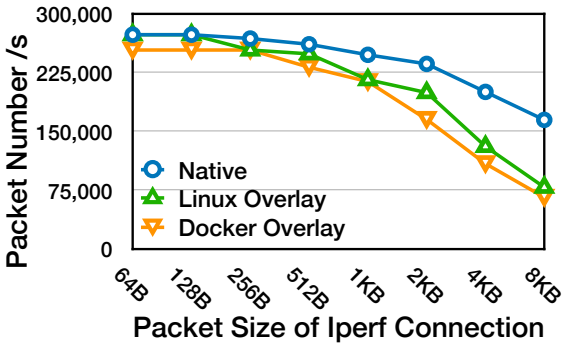

Figure 2.7: TCP packet processing rate with varying packet sizes.

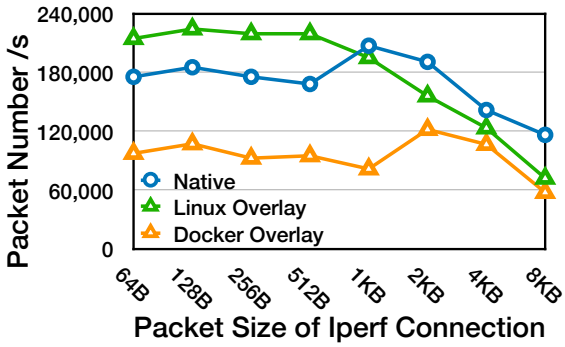

Figure 2.8: UDP packet processing rate with varying packet sizes.

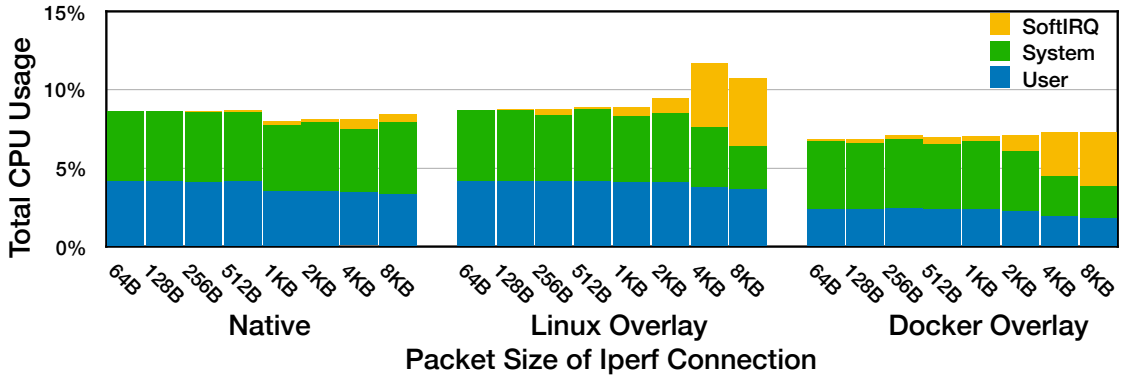

Figure 2.9: CPU usage breakdown on receiver side with varying packet sizes (TCP).

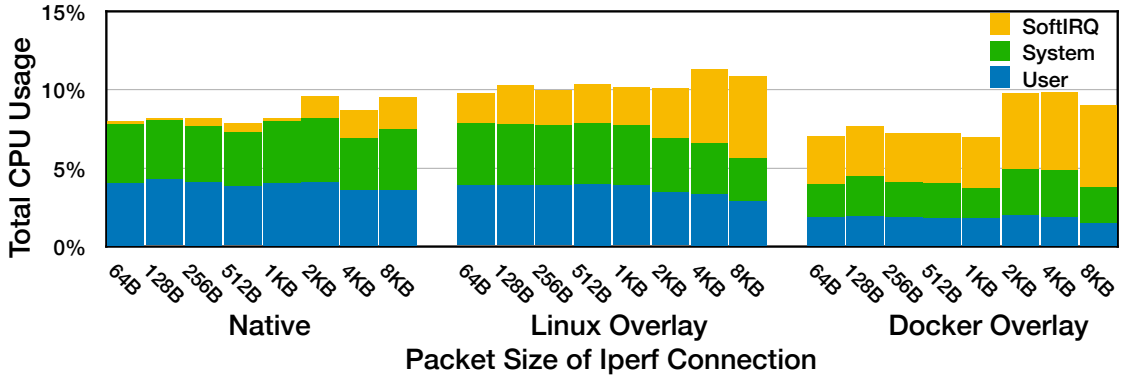

Figure 2.10: CPU usage breakdown on receiver side with varying packet sizes (UDP).

with small packet sizes (64 bytes to 1 KB) than the native under TCP in terms of packet processing rate; the gap becomes wider as the packet size increases. Further, as shown in Figure 2.9, the Docker overlay consumes less CPU due to lower packet processing rate [3].

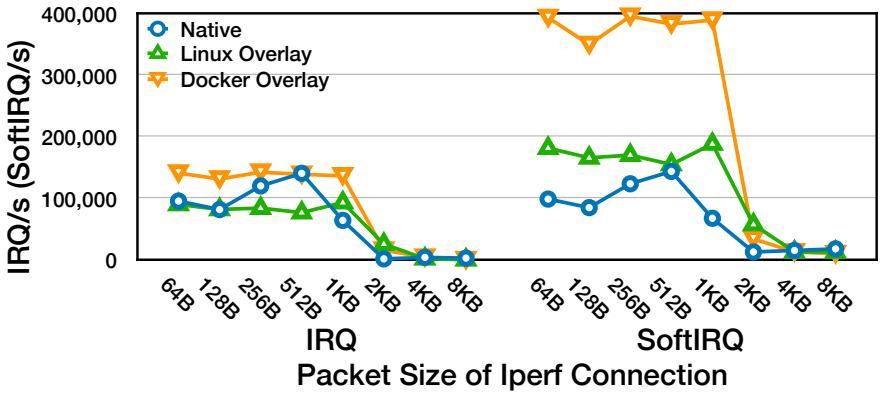

Figure 2.11: Interrupt number with varying packet sizes (UDP).

The more significant inefficiency is observed in the UDP case: In Figure 2.8, we observe that the Docker overlay achieves as low as 50% packet processing rate of that in the native case with lower CPU usage (Figure 2.10). The Linux overlay case performs better than the Docker overlay but still worse than the native. Correspondingly, we observe that the IRQ number increases dramatically in the Docker overlay case — *10x* of that in the TCP case. In addition, much more softirqs are observed in Figure 2.11, $\sim$*3x* of the IRQs. It is because again, one IRQ can trigger (at most) three softirqs in the Docker overlay case. Note that, multiple softirqs can"merge" within one softirq, as long as they are processed in a timely manner (i.e., all pro-

[3]The Docker overlay is more CPU efficient than the Linux overlay under small packet sizes, as (we observed that) the kernel CPU scheduler intends to put the user-space iperf processes on the same core — that also performs kernel-level packet processing — more often in the Docker overlay case.

cessed under the context of one softirq and counted once). Notice that, more softirqs indicate that either the IRQ number is large or the process of softirqs is frequently interrupted (multiple softirqs cannot merge) — the Docker overlay case falls in the latter category.

2.4 Discussion

By presenting our observations in container overlay networks, we are looking to receive feedback that can gauge the importance of these observed bottlenecks considering real cloud containerized applications. We are aware of that there is a large body of work addressing the inefficient network packet processing issue with either optimizing existing operating systems (OS), or renovating OSes with a clean-slate design, or completely bypassing the OSes with a user-space approach. However, in our work, we aim to first have a thorough understanding about the inefficiencies of conventional OSes particularly for container overlay networks. With this, we plan to generate discussions about whether we should keep improving the conventional kernel network stack following an evolutionary concept by retrofitting existing OSes with the new technology for better adoptability and compatibility.

2.5 Conclusion

We have presented the performance study of container overlay networks on a multi-core system with high-speed network devices, and identified three critical parallelization bottlenecks in the kernel network stack which prevent overlay networks from scaling: (1) the kernel does not provide per-packet level parallelization preventing a single container flow from achieving high network throughput; (2) the kernel does not efficiently handle various packet processing tasks preventing multiple container flows

from easily saturating a 40 Gbps network link; and (3) the above two parallelization bottlenecks become more severe for small packets, as the kernel fails to handle a large number of interrupts which disrupts the overall system efficiency.

These parallelization bottlenecks urge us to develop a more efficient kernel network stack for overlay networks by considering the following several questions: (1) Is it feasible to provide packet-level parallelization for a single network flow? Though probably not necessary in the native case, it becomes imperative in the overlay networks as the achieved throughput of a single flow is still very low (limited by a single CPU core). (2) How can the kernel perform a better isolation among multiple flows especially for efficiently utilizing shared hardware resources (e.g., CPU caches and memory bandwidth). This is particularly important as one server can host tens or even hundreds of light-weight containers. It becomes more challenging to handle small packets under overlay networks. (3) Can the packets be further coalesced with optimized network path for reduced interrupts and context switches?

CHAPTER 3

PARALLELIZING PACKET PROCESSING IN CONTAINER OVERLAY NETWORKS

Container networking, which provides connectivity among containers on multiple hosts, is crucial to building and scaling container-based microservices. While overlay networks are widely adopted in production systems, they cause significant performance degradation in both throughput and latency compared to physical networks. This chapter seeks to understand the bottlenecks of in-kernel networking when running container overlay networks. Through profiling and code analysis, we find that a prolonged data path, due to packet transformation in overlay networks, is the culprit of performance loss. Furthermore, existing scaling techniques in the Linux network stack are ineffective for parallelizing the prolonged data path of a single network flow.

We propose Falcon, a fast and balanced container networking approach to scale the packet processing pipeline in overlay networks. Falcon pipelines software interrupts associated with different network devices of a single flow on multiple cores, thereby preventing execution serialization of excessive software interrupts from overloading a single core. Falcon further supports multiple network flows by effectively multiplexing and balancing software interrupts of different flows among available cores. We have developed a prototype of Falcon in Linux. Our evaluation with both micro-benchmarks and real-world applications demonstrates the effectiveness of Falcon, with significantly improved performance (by 300% for web serving) and reduced tail latency (by 53% for data caching).

3.1 Introduction

Due to its high performance [33, 34], low overhead [35, 15], and widespread community support [36], container technology has increasingly been adopted in both private data centers and public clouds. A recent report from Datadog [37] has revealed that customers quintupled the number of containers in their first nine-month container adoption. Google deploys containers in its cluster management and is reported to launch about 7,000 containers every second in its search service [4]. With containers, applications can be automatically and dynamically deployed across a cluster of physical or virtual machines (VMs) with orchestration tools, such as Apache Mesos [5], Kubernetes [6], and Docker Swarm [7].

Container networks provide connectivity to distributed applications and are critical to building large-scale, container-based services. Overlay networks, e.g., Flannel [8], Weave [9], Calico [10] and Docker overlay [11], are widely adopted in container orchestrators [5, 6, 7]. Compared to other communication modes, overlay networks allow each container to have its own network namespace and private IP address independent from the host network. In overlay networks, packets must be transformed from private IP address to public (host) IP address before transmission, and vice versa during reception. While network virtualization offers flexibility to configure private networks without increasing the complexity of host network management, packet transformation imposes significant performance overhead. Compared to a physical network, container overlay networks can incur drastic throughput loss and suffer an order of magnitude longer tail latency [15, 38, 39, 14, 1].

The overhead of container overlay networks is largely due to a prolonged data path in packet processing. Overlay packets have to traverse the private overlay network stack and the host stack [14] for both packet transmission and reception. For instance, in a virtual extensible LAN (VXLAN) overlay, packets must go through a

`VXLAN` device for IP transformation, i.e., adding or removing host network headers during transmission or reception, a virtual bridge for packet forwarding between private and host stacks, and a virtual network device (`veth`) for gating a container's private network. The inclusion of multiple stages (devices) in the packet processing pipeline prolongs the critical path of a *single* network flow, which can only be processed on a single core.

The existing mechanisms for parallelizing packet processing, such as Receive Packet Steering (RPS) [27], focus on distributing multiple flows (packets with different IPs or ports) onto separate cores, thereby not effective for accelerating a single flow. The prolonged data path inevitably adds delay to packet processing and causes spikes in latency and significant throughput drop if computation overloads a core. To shorten the data path, the state-of-the-art seeks to either eliminate packet transformation from the network stack [14] or offload the entire virtual switches and packet transformation to the NIC hardware[40]. Though the performance of such software-bypassing or hardware-offloading network is improved (close to the native), these approaches undermine the flexibility in cloud management with limited support and/or accessibility. For example, Slim [14] does not apply to connection-less protocols, while advanced hardware offloading is only available in high-end hardware [40].

This chapter investigates how and to what extent the conventional network stack can be optimized for overlay networks. We seek to preserve the current design of overlay networks, i.e., constructing the overlay using the existing building blocks, such as virtual switches and virtual network devices, and realizing network virtualization through packet transformation. This helps to retain and support the existing network and security policies, and IT tools. Through comprehensive profiling and analysis, we identify previously unexploited parallelism within a *single* flow in overlay networks: Overlay packets travel multiple devices across the network stack and the processing at

each device is handled by a separate software interrupt (softirq); while the overhead of container overlay networks is due to *excessive* softirqs of one flow overloading a single core, the softirqs are *asynchronously* executed and their invocations can be interleaved. This discovery opens up new opportunities for parallelizing softirq execution in a single flow with multiple cores.

We design and develop FALCON (fast and balanced container networking) — a novel approach to parallelize the data path of a single flow and balance network processing pipelines of multiple flows in overlay networks. FALCON leverages multiple cores to process packets of a single flow at different network devices via a new hashing mechanism: It takes not only flow but also network device information into consideration, thus being able to distinguish packet processing stages associated with distinct network devices. FALCON uses in-kernel stage transition functions to move packets of a flow among multiple cores in sequence as they traverse overlay network devices, preserving the dependencies in the packet processing pipeline (i.e., no out-of-order delivery). Furthermore, to exploit parallelism within a heavy-weight network device that overloads a single core, FALCON enables a softirq splitting mechanism that splits the processing of a heavy-weight network device (at the function level), into multiple smaller tasks that can be executed on separate cores. Last, FALCON devises a dynamic balancing mechanism to effectively multiplex softirqs of multiple flows in a multi-core system for efficient interrupt processing.

Though FALCON piplelines the processing stages of a packet on multiple cores, it does *not* require packet copying between these cores. Our experimental results show that the performance gain due to parallelization significantly outweighs the cost of loss of locality. To summarize, this chapter has made the following contributions:

- We perform a comprehensive study of the performance of container overlay networks and identify the main bottleneck to be the serialization of a large number of softirqs on a single core.
- We design and implement FALCON that parallelizes the prolonged data path for a single flow in overlay networks. Unlike existing approaches that only parallelize softirqs at packet reception, FALCON allows softirqs to be parallelized at any stage of the processing pipeline.
- We evaluate the effectiveness of FALCON with both micro and real-world applications. Our results show that FALCON can significantly improve throughput (e.g., up to 300% for web serving) and reduce latency (e.g., up to 53% for data caching).

3.2 Background and Motivation

In this section, we first describe the process of packet processing in the OS kernel. Then, we examine the performance bottleneck of container overlay networks. Without loss of generality, we focus on packet reception in the Linux kernel because reception is in general harder than transmission and incurs greater overhead in overlay networks. Furthermore, packet reception presents the parallelism that can be exploited to accelerate overlay networks.

3.2.1 Background

In-kernel packet processing. Packet processing in commodity OSes is a pipeline traversing the network interface controller (NIC), the kernel space and the user space, as shown in Figure 3.1. Take packet reception for example, when a packet arrives at the NIC, it is first copied (e.g., via DMA) to the device buffer and triggers a hardware interrupt. Then the OS responds to the interrupt and transfers the packet through

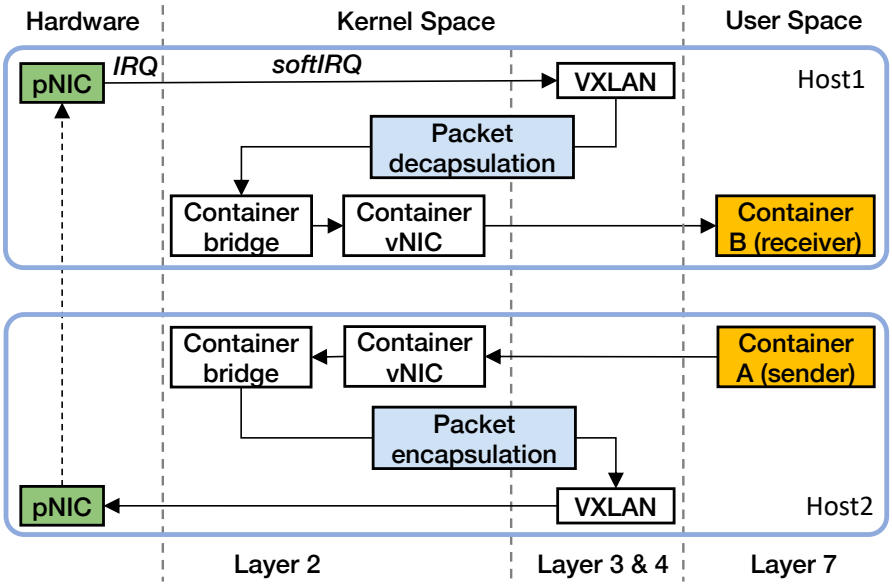

Figure 3.1: Illustration of container overlay networks.

the receiving path in the kernel. Packet processing is divided into the top half and bottom half. The top half runs in the hardware interrupt context. It simply marks that the packet arrives at the kernel buffer and invokes the bottom half, which is typically in the form of a software interrupt, softirq. The softirq handler — the main routine to transfer packets along the protocol stack — is later scheduled by the kernel at an appropriate time. After being processed by various protocol layers, the packet is finally copied to a user-space buffer and delivered to the applications listening on the socket.

Container overlay network. In the pursuit of management flexibility, virtualized networks are widely adopted in virtualized servers to present logical network views to end applications. Overlay network is a common way to virtualize container networks. As an example in Figure 3.1, in a container overlay network (e.g., VXLAN), when a packet is sent from container A to container B, the overlay layer (layer 4) of container A first looks up the IP address of the destination host where container B resides — from a distributed key-value store which maintains the mapping between private IP

addresses of containers and the public IP addresses of their hosts. The overlay network then encapsulates the packet in a new packet with the destination host IP address and places the original packet as the payload. This process is called *packet encapsulation*. Once the encapsulated packet arrives at the destination host, the overlay layer of container B decapsulates the received packet to recover the original packet and finally delivers it to container B identified by its private IP address. This process is called *packet decapsulation*. In addition to the overlay networks, the container network also involves additional virtualized network devices, such as bridges, virtual Ethernet ports (vNIC), routers, etc., to support the connectivity of containers across multiple hosts. Compared to the native network, container overlay network is more complex with a longer data path.

Interrupts on multi-core machines. The above network packet processing is underpinned by two types of interrupts: hardware interrupts (hardirqs) and software interrupts (softirqs). On the one hand, like any I/O devices, a physical NIC interacts with the OS mainly through hardirqs. A physical NIC with one traffic queue is assigned with an IRQ number during the OS boot time; hardirqs triggered by this NIC traffic queue can only be processed on *one* CPU core at a time in an IRQ context of the kernel (i.e., the IRQ handler). To leverage multi-core architecture, a modern NIC can have multiple traffic queues each with a different IRQ number and thus interacting with a separate CPU core. On the other hand, an OS defines various types of softirqs, which can be processed on any CPU cores. Softirqs are usually raised when an IRQ handler exits and processed on the *same* core (as the IRQ handler) by the softirq handler either immediately (right after the IRQ handler) or asynchronously (at an appropriate time later). Typically, the hardirq handler is designed to be simple and small, and runs with hardware interrupts on the same core

disabled (cannot be preempted), while the softirq handler processes most of the work in the network protocol stack and can be preempted.

Packet steering is a technique that leverages multiple cores to accelerate packet processing. Receive side scaling (RSS) [26] steers packets from different flows to a separate receive queue on a multi-queue NIC, which later can be processed by separate CPUs. While RSS scales packet processing by mapping hardirqs to separate CPUs, receive packet steering (RPS) [27] is a software implementation of RSS and balances softirqs. Both RSS and RPS calculate a flow hash based on the packet's IP address and port and use the hash to determine the CPU on which to dispatch the interrupts.

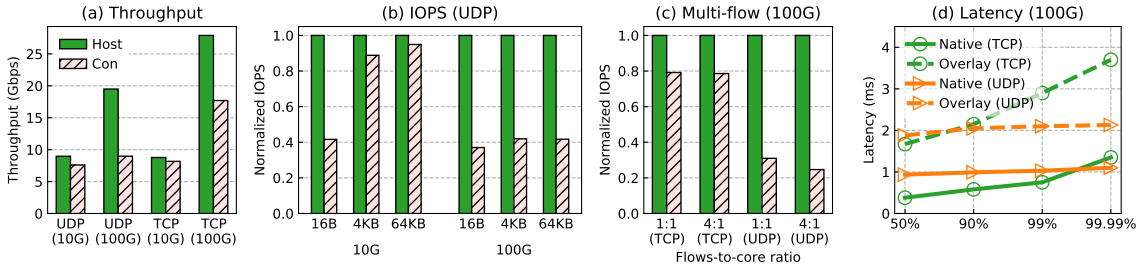

Figure 3.2: Performance comparison of container overlay networks and native physical networks.

3.2.2 **Motivation**

Experimental settings. We evaluated the throughput and latency of the VXLAN overlay network between a pair of client and server machines and studied how its performance is different from the native host network. The machines were connected with two types of NICs over direct links: Intel X550T 10-Gigabit and Mellanox ConnectX-5 EN 100-Gigabit Ethernet adapters. Both the client and server had abundant CPU and memory resources.

Single-flow throughput. Figure 3.2 depicts the performance loss due to the overlay network in various settings. Figure 3.2 (a) shows the comparison between overlay and host networks in a throughput stress test. We used *sockperf* [41] with large packets (64 KB for both TCP and UDP) using a single flow. To determine the maximum achievable throughput, we kept increasing the sending rate until received packet rate plateaued and packet drop occurred. While the overlay network achieved near-native throughput in the slower 10 Gbps network, which is similar to the findings in Slim [14], it incurred a large performance penalty in the faster 100 Gbps network for both UDP and TCP workloads by 53% and 47%, respectively. The results suggest that overlay networks impose significant per-packet overhead that contributes to throughput loss but the issue is often overlooked when link bandwidth is the bottleneck and limits packet rate.

Single-flow packet rate. Figure 3.2 (b) shows packet rates (IOs per second) under different packet sizes for UDP traffic. When the packet size was small, the network stack's ability to handle a large number of packets limited the packet rate and led to the largest performance gap between overlay and host networks while link bandwidth was no longer the bottleneck. As packet size increased, the gap narrowed. But for the faster 100 Gbps Ethernet, the performance degradation due to overlay networks had always been significant. Tests on TCP workloads showed a similar trend.

Multi-flow packet rate. Next, we show that the prolonged data path in a single flow may have a greater impact on multi-flow performance. Both the host and overlay network had packet steering technique receive packet steering (RPS) enabled. Figure 3.2 (c) shows multi-flow packet rate with two flow-to-core ratios. A 1:1 ratio indicates that there are sufficient cores and each flow (e.g., a TCP connection) can be processed by a dedicated core. Otherwise, with a higher ratio, e.g., 4:1, multiple flows are mapped to the same core. The latter resembles a more realistic scenario

wherein a server may serve hundreds, if not thousands, of connections or flows. The packet size was 4 KB.

A notable finding is that overlay networks incurred greater throughput loss in multi-flow tests compared to that in single-flow tests, even in tests with a $1:1$ flow-to-core ratio. Packet steering techniques use consistent hashing to map packets to different cores. When collisions occur, multiple flows may be placed on the same core even idle cores are available, causing imbalance in flow distribution. Since individual flows become more expensive in overlay networks, multi-flow workloads could suffer a greater performance degradation in the presence of load imbalance. Furthermore, as flow-to-core ratio increased, throughput loss further exacerbated.

Latency. As shown in Figure 3.2 (d), it is expected that given the prolonged data path, overlay networks incur higher latency than the native host network in both UDP and TCP workloads. The figure suggests up to 2x and 5x latency hike for UDP and TCP, respectively.

Summary. Container overlay networks incur significant performance loss in both throughput and latency. The performance penalty rises with the speed of the underlying network and packet rate. In what follows, we analyze the root causes of overlay-induced performance degradation.

3.3 Root Cause Analysis

3.3.1 Prolonged Data Path

We draw the call graph of packet reception in the Linux kernel using `perf` and `flamegraph` [42] and analyze the control and data paths in the host and overlay networks. As Figure 3.3 illustrates, packet reception in an overlay network involves

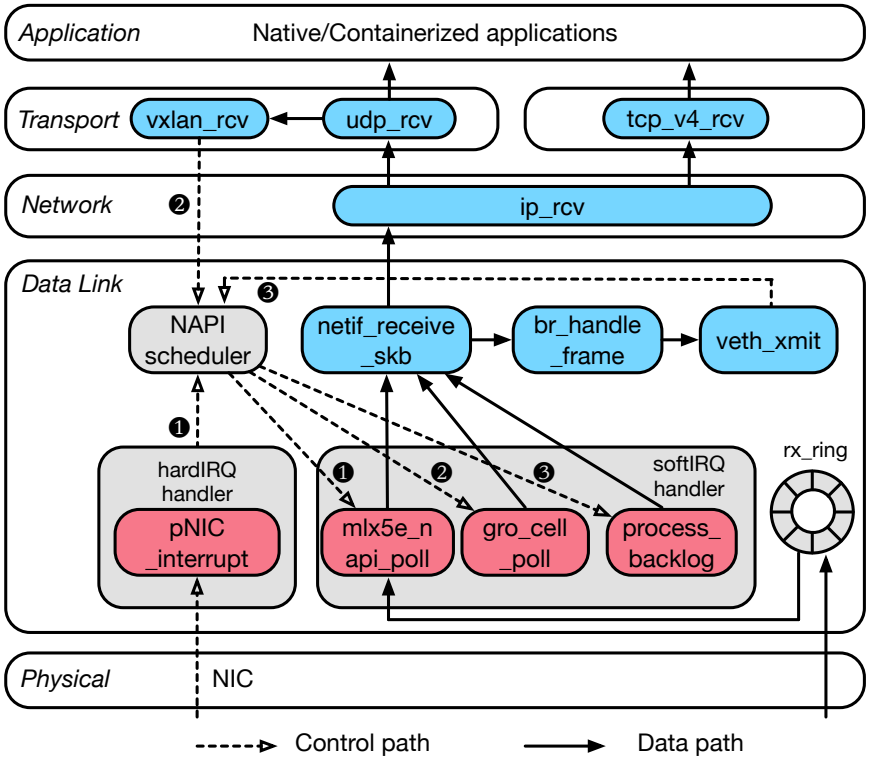

Figure 3.3: Packet reception in container overlay networks.

multiple stages. The numbered steps are the invocation of hardware or software interrupts on different network devices (❶: physical NIC, ❷: `VXLAN`, ❸: `veth`).

In host network, upon packet arrival, the physical NIC raises a hardirq and copies the packet into a receiving ring buffer (`rx_ring`) in the kernel. In response to the hardirq, the IRQ handler (`pNIC_interrupt`) is immediately executed (❶), during which it raises softirqs on the same CPU it is running. Later, the softirq handler (`net_rx_action`) is invoked by the Linux NAPI scheduler; it traverses the polling list and calls the polling function provided by each network device to process these softirqs. In the native network, only one polling function – physical NIC (`mlx5e_napi_poll`) (❶) is needed. It polls packets from the ring buffer and passes them to the entry function of the kernel network stack (`netif_receive_skb`). After

processed by each kernel stack, packets are finally copied to the socket buffer and received by userspace applications. Note that the entire packet processing is completed in **one** single softirq.

In comparison, packet processing in an overlay network is more complex, requiring to traverse multiple network devices. The initial processing in an overlay shares step ❶ with the physical network until packets reach the transport layer. The UDP layer receive function `udp_rcv` invokes the VXLAN receive routine `vxlan_rcv` if a packet is found to contain an inner packet with a private IP. `vxlan_rcv` decapsulates the packet by removing the outer VXLAN header, inserts it at the tail of the receive queue of the VXLAN device, and raises another `NET_RX_SOFTIRQ` softirq (step ❷). The softirq uses the VXLAN device's polling function `gro_cell_poll` to pass packets to the upper network stack.

Furthermore, containers are usually connected to the host network via a bridge device (e.g., Linux bridge or Open vSwitch [13]) and a pair of virtual Ethernet ports on device `veth`. One `veth` port attaches to the network bridge while the other attaches to the container, as a gateway to the container's private network stack. Thus, the packets (passed by `gro_cell_poll`) need to be further processed by the bridge processing function (`br_handle_frame`) and the `veth` processing function (`veth_xmit`). More specifically, the `veth` device on the bridge side inserts the packets to a per-CPU receiving queue (`input_pkt_queue`) and meanwhile raises a third softirq (`NET_RX_SOFTIRQ`) (step ❸). Since `veth` is not a NAPI device, the default poll function `process_backlog` is used to pass packets to the upper protocol stack. Therefore, packet processing in a container overlay network involves three network devices with the execution of **three** softirqs.

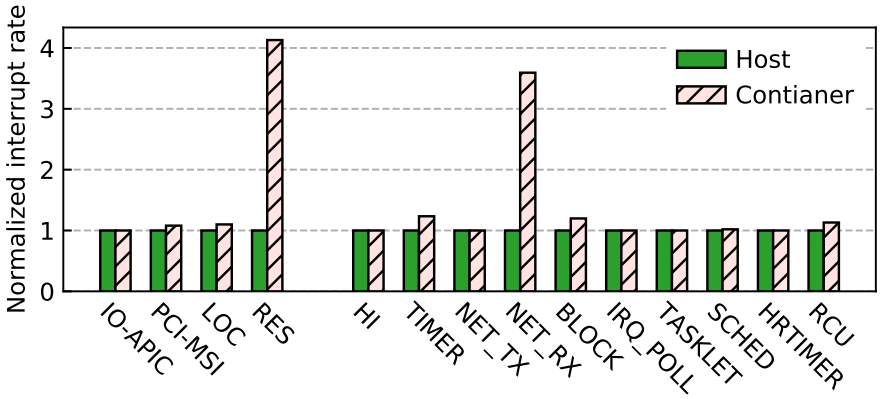

Figure 3.4: Comparison of hardware and software interrupt rates in the native and container overlay networks.

3.3.2 **Excessive, Expensive, and Serialized Softirqs**

Call graph analysis suggests that overlay networks invoke more softirqs than the native network does. Figure 3.4 confirms that the overlay network triggers an excessive number of the `RES` and `NET_RX` interrupts. `NET_RX` is the softirq that handles packet reception. The number of `NET_RX` in the overlay network was 3.6x that of the native network. The results confirm our call graph analysis that overlay networks invoke three times of softirqs than the native network.

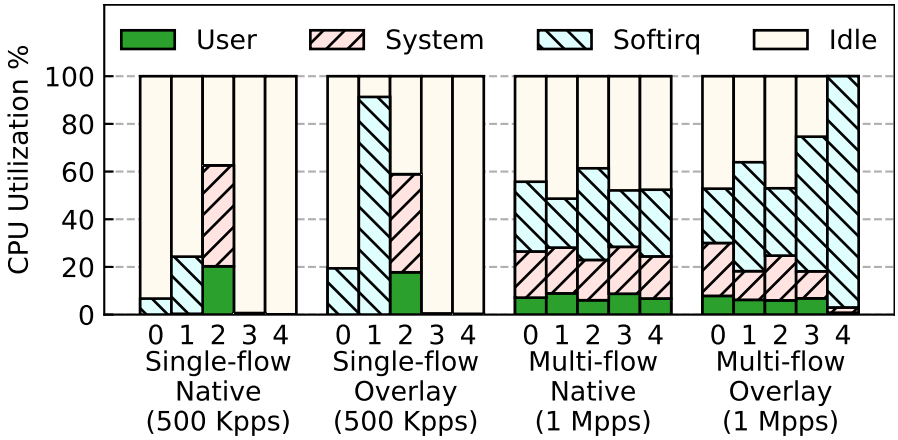

Figure 3.5: Serialization of softIRQs and load imbalance.

Our investigation on RES – the rescheduling interrupt, further reveals that there exists significant load imbalance among multiple cores when processing overlay packets. RES is an inter-processor interrupt (IPI) raised by the CPU scheduler attempting to spread load across multiple cores. Figure 3.5 shows the CPU utilization in host and overlay networks for single-flow and multi-flow tests in the 100 Gbps Ethernet. The workloads were sockperf UDP tests with fixed sending rates. Note that the sending rates were carefully set to keep the server reasonably busy without overloading it. This allows for a fair comparison of their CPU utilization facing the same workload. The figure shows that overlay network incurred much higher CPU utilization compared to the native network, mostly on softirqs. Moreover, most softirq processing was stacked on a single core. (e.g., core 1 in the single-flow overlay test). The serialization of softirq execution can quickly become the bottleneck as traffic intensity ramps up. The multi-flow tests confirmed softirq serialization — the OS was unable to use more than 5 cores, i.e., the number of flows, for packet processing. The overlay network also exhibited considerable imbalance in core utilization due to possible hash collisions in RPS, which explains the high number of RES interrupts trying to perform load balancing.

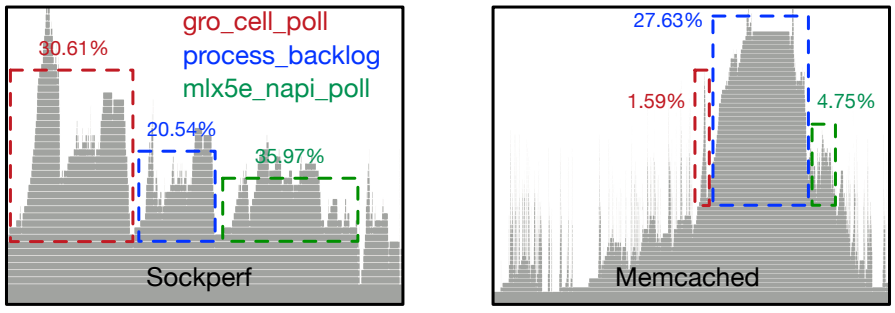

Figure 3.6: Flamegraphs of Sockperf and Memcached.

Not only are there more softirqs in overlay networks, some of them become more expensive than that in the native network. Figure 3.6 shows the flamegraphs of function invocation in sockperf and memcached. The former is a micro-benchmark that has only one type of packets with uniform sizes while the latter is a realistic application that includes a mixture of TCP and UDP packets with different sizes. The flamegraphs demonstrate that for workloads with simple packet types the overhead of overlay networks is manifested by additional, relatively equally weighted softirqs. In contrast, certain softirqs become particularly expensive and dominate overlay overhead in realistic workloads.

3.3.3 Lack of Single-Flow Parallelization

Packet steering techniques seek to reduce the data-plane overhead via *inter-flow* parallelization. However, these mechanisms are not effective for parallelizing a single flow as all packets from the same flow would have the same hash value and thus are directed to the same CPU. As shown in Figure 3.5 (left, single-flow tests), although packet steering (i.e., RSS and RPS) does help spread softirqs from a single flow to two cores, which agrees with the results showing packet steering improves TCP throughput for a single connection in Slim [14], most of softirq processing is still stacked on one core. The reason is that packet steering takes effect early in the packet processing pipeline and does help separate softirq processing from the rest of data path, such as hardirqs, copying packets to the user space, and application threads. Unfortunately, there is a lack of mechanisms to further parallelize the softirq processing from the same flow.

There are two challenges in scaling a single flow: 1) Simply dispatching packets of the same flow to multiple CPUs for processing may cause out-of-order delivery as different CPUs may not have a uniform processing speed. 2) For a single flow involving

multiple stages, as is in the overlay network, different stages have little parallelism to exploit due to inter-stage dependency. Hence, performance improvement can only be attained by exploiting packet-level parallelism.

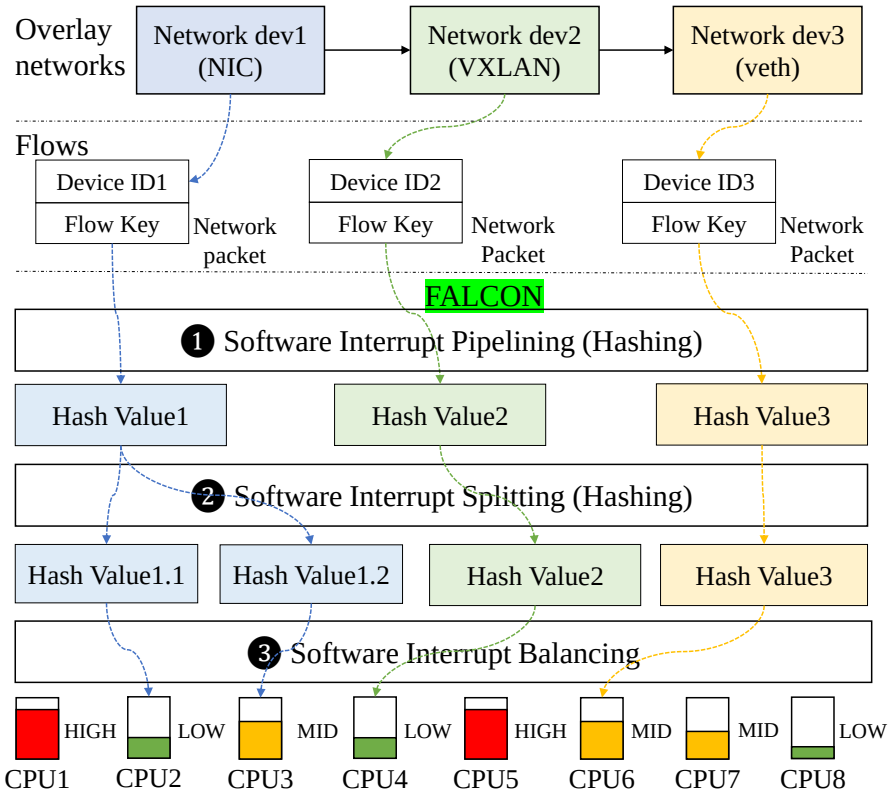

Figure 3.7: Architecture of FALCON.

3.4 Design

The previous section suggests that, due to the lack of single-flow parallelization, the execution of excessive softirqs from multiple network devices in container overlay networks can easily overload a single CPU core, preventing a single flow from achieving high bandwidth and resulting in long tail latency. To address this issue, we design and develop FALCON with the key idea as follows: Instead of processing all

softirqs of a flow on a single core, FALCON pipelines softirqs associated with different devices on *separate* cores, while still preserving packet processing dependencies among these devices and in-order processing on each device. To realize this idea, FALCON incorporates three key components, software interrupt pipelining, software interrupt splitting, and dynamic load balancing (in Figure 3.7), as detailed as follows.

3.4.1 Software Interrupt Pipelining

Inspired by RPS [27], which dispatches different network flows onto multiple cores via a hashing mechanism, FALCON aims to dispatch the different packet processing stages (associated with different network devices) of a single flow onto separate cores. This way, FALCON exploits the parallelism of a flow's multiple processing stages by leveraging multiple cores, while still preserving its processing dependencies — packets are processed by network devices sequentially as they traverse overlay network stacks. Furthermore, as for each stage, packets of the same flow are processed on *one* dedicated core, FALCON avoids "out-of-order" delivery.

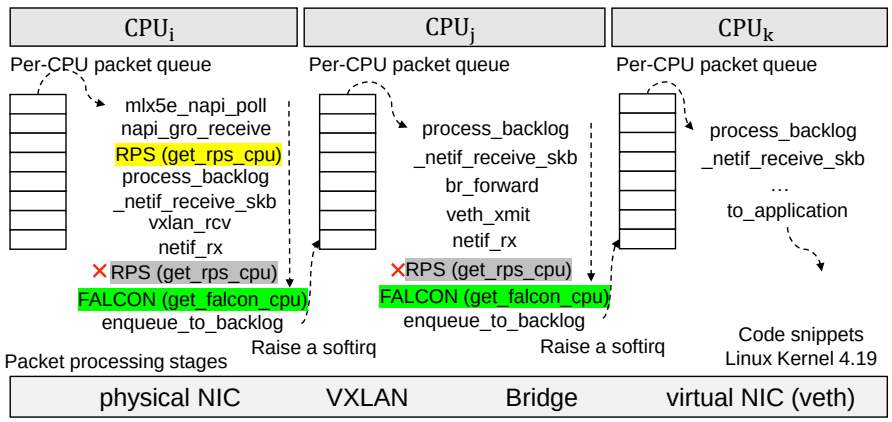

Figure 3.8: FALCON pipelines software interrupts of a single flow by leveraging stage transition functions.

Unfortunately, we find that the existing hashing mechanism used by RPS cannot distinguish packet processing stages associated with different network devices (e.g., NIC, VXLAN, bridge, and veth in Figure 3.3), as it simply takes packet information as input without considering device information. Specifically, the existing hash mechanism in RPS performs the hash calculation upon a network flow key (`flow_keys`) — a data structure composed of a packet's source and destination IP addresses, protocol, ports, tags, and other metadata needed to identify a network flow. The calculated hash value is used to determine the core on which the packet will be processed. Yet, since the hash calculation does not include device information, all stages of the packets of a single flow are executed on the same core. As illustrated in Figure 3.8, though the RPS function is invoked multiple times along the network path, only the first RPS (on $\texttt{CPU}_\texttt{i}$) takes effect (i.e., selecting a new CPU core based on the hash value), while the following RPS (e.g., on $\texttt{CPU}_\texttt{i}$ and on $\texttt{CPU}_\texttt{j}$) generate the same hash value for the packets of the same flow.

A natural way to distinguish different processing stages of a single flow is to involve additional device information for the hash calculation: We notice that, when a packet is sent or received by a new network device, the device pointer (`dev`) in the packet's data structure (`sk_buff`) will be updated and pointed to that device. Therefore, we could involve the index information of network devices (e.g., `dev`→`ifindex`) in the hash calculation, which would generate distinct hash values for different network devices. However, simply reusing RPS functions that are statically located along the existing network processing path may unnecessarily (and inappropriately) split the processing of one network device into fragmented pieces distributed on separate cores — as we can see in Figure 3.8, two RPS functions are involved along the processing path of the first network device (i.e., `pNIC`).

Instead, FALCON develops a new approach to separate distinct network processing stages via *stage transition functions*. We find that certain functions in the kernel network stack act as stage transition functions — instead of continuing the processing of a packet, they enqueue the packet into a device queue that will be processed later. The `netif_rx` function is such an example as shown in Figure 3.8, which by default enqueues a packet to a device queue. The packet will be retrieved from the queue and processed later on the same core. These stage transition functions are originally designed to multiplex processings of multiple packets (from multiple flows) on the *same* core, while FALCON re-purposes them for a multi-core usage: At the end of each device processing [1], FALCON reuses (or inserts) a stage transition function (e.g., `netif_rx`) to enqueue the packet into a target CPU's per-CPU packet queue. To select the target CPU, FALCON employs a CPU-selection function, which returns a CPU based on the hash value calculated upon both the flow information (e.g., `flow_keys`) and device information (e.g., `ifindex`) — i.e., distinct hash values for different network devices given the same flow. Finally, FALCON raises a softirq on the target CPU for processing the packet at an appropriate time.

With stage transition functions, FALCON can leverage a multi-core system to freely pipeline a flow's multiple processing stages on separate CPU cores — the packets of a single flow can be associated with nonidentical cores for processing when they enter distinct network devices. FALCON's design has the following advantages: 1) It does not require modifications of existing network stack data structures (e.g., `sk_buff` and `flow_keys`) for hash calculation, making FALCON portable to different kernel versions (e.g., we have implemented FALCON in kernel 4.19 and easily ported it to kernel 5.4); 2) Since FALCON uses stage transition functions (instead of reusing RPS)

[1] FALCON can also stack multiple devices in one processing stage, aiming to evenly split the network processing load on multiple cores.

for separation of network processing, it can coexist with existing scaling techniques like RPS/RSS.

3.4.2 Software Interrupt Splitting

Though it makes intuitive sense to separate network processing stages at *per-device* granularity, our analysis of the Linux kernel (from version 4.19 to 5.4) and the performance of TCP and UDP with various packet sizes reveal that, a finer-grained approach to split network processing stages is needed under certain circumstances.

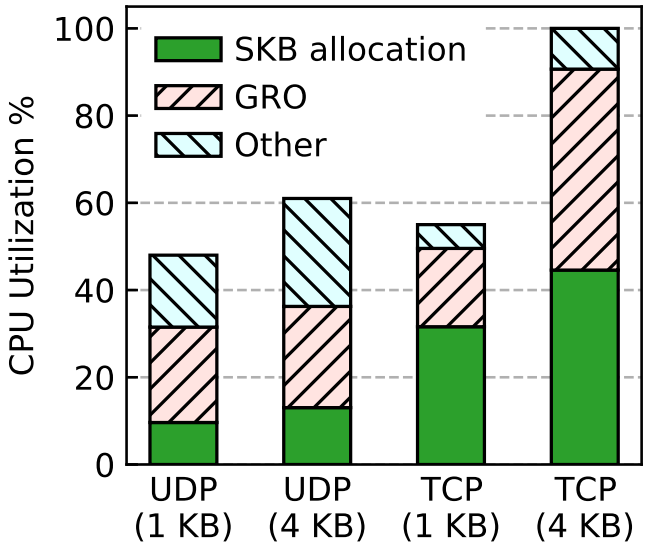

Figure 3.9: CPU% of the first stage packet processing.

As plotted in Figure 3.9, under the TCP case with a large packet size (e.g., 4 KB), the first stage of Falcon (associated with the physical NIC) easily takes up 100% of a single CPU core and becomes the new bottleneck. Upon deep investigation, we identify that two functions (`skb_allocation` and `napi_gro_receive`) are the culprits, with each contributing around 45% of CPU usage. However, such a case does not exist under UDP or TCP with small packets (e.g., 1 KB), where the first stage

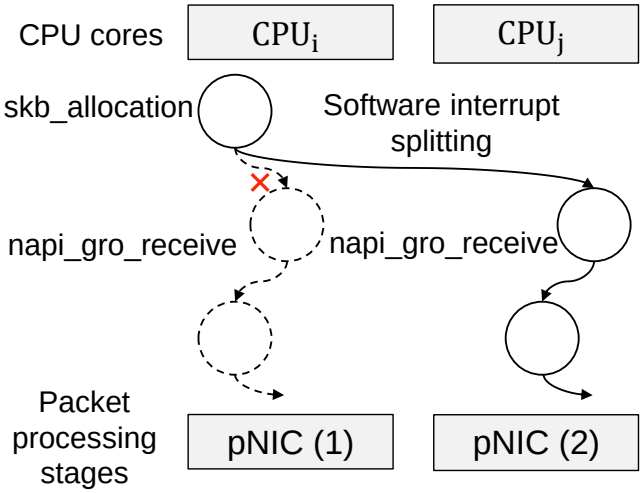

Figure 3.10: Software interrupt splitting.

does not saturate a single core. It is because, the GRO [2] function (`napi_gro_receive`) is heavily involved in processing TCP flows with a large packet size, while it merely takes effect for UDP flows or TCP flows with a small packet size. This issue – the processing of one network device overloads a single CPU core – could commonly exist, as the Linux network stack is designed to be flexible enough that allows arbitrary network devices or modules to be "hooked" on demand along the network path, such as container's overlay device (`VXLAN`), traffic encryption [43], profiling [44], in-kernel software switches [13], and many network functions [45, 46, 47].

To further exploit parallelism within a "heavy-weight" network device that overloads a single core, FALCON enables a softirq splitting mechanism: It separates the processing functions associated with the network device onto multiple cores by inserting stage transition functions right before the function(s) to be offloaded. In the example of Figure 3.10, to offload the CPU-intensive GRO function (e.g., under TCP with 4 KB packet size), FALCON inserts a transition function (i.e., `netif_rx`) before

[2] The generic receive offload (GRO) function reassembles small packets into larger ones to reduce per-packet processing overheads.

the GRO function. Meanwhile, a softirq is raised on the target core, where the GRO function is offloaded. By doing this, FALCON splits the original one softirq into two, with each for a half processing of the associated network device (e.g., `pNIC1` and `pNIC2` in Figure 3.10).

Note that, FALCON's softirq splitting mechanism is general in that FALCON can arbitrarily split the processing of any network device, at the function level, into multiple smaller tasks, which can be parallelized on multiple cores. However, it should be applied with discretion, as splitting does incur additional overhead, such as queuing delays, and it could offset the performance benefit from the parallelism. In practice, FALCON only applies software interrupt splitting to a network device that fully overloads a CPU core [3].

3.4.3 Software Interrupt Balancing

The use of stage transition functions is a generic approach to resolve the bottleneck of overlay networks by parallelizing softirq processing of a single flow as well as breaking expensive softirqs into multiple smaller softirqs. Challenges remain in how to effectively and efficiently balance the softriqs to exploit hardware parallelism and avoid creating new bottlenecks. First, the kernel network stack may coalesce the processing of packets from different flows in the same softirq to amortize the overhead of softirq invocation. Thus, softirq balancing must be performed on a per-packet basis as downstream softirqs from different flows should be sent to different cores. Since packet latency is in the range of tens of to a few hundreds of microseconds, the cost to evenly distribute softirqs should not add much delay to the latency. Second, load balancing relies critically on load measurements to determine where softirqs should

[3] FALCON statically splits functions of a heavy-weight network device, via offline profiling. Yet, we note that a dynamic method is more desired, which is the subject of our ongoing investigations.

be migrated from and to. However, per-packet softirq balancing on individual cores lacks timely and accurate information on system-wide load, thereby likely to create new bottlenecks. A previous lightly loaded core may become a hotspot if many flows dispatch their softirqs to this core and CPU load may not be updated until the burst of softirqs has been processed on this core.

The fundamental challenge is the gap between fine-grained, distributed, per-packet balancing and the complexity of achieving global load balance. To overcome it, FALCON devises a dynamic softirq balancing algorithm that 1) prevents overloading any core and 2) maintains a reasonably good balance across cores 3) at a low cost. As shown in Algorithm 1, the dynamic balancing algorithm centers on two designs. First, FALCON is enabled only when there are sufficient CPU resources to parallelize individual network flows otherwise all softirqs stay on the original core (line 6–9). FALCON monitors system-wide CPU utilization and switches softirq pipelining and splitting on and off according to `FALCON_LOAD_THRESHOLD`. Second, FALCON employs a *two-choice* algorithm for balancing softirqs: 1) it first computes a hash on the device ID and the flow key to uniquely select a CPU for processing a softirq (line 19–20). Given the nature of hashing, the first choice is essentially a uniformly random CPU in the FALCON CPU set. This helps evenly spread softirqs across CPUs without quantitatively comparing their loads. If the first selected CPU is busy, FALCON performs double hashing to pick up another CPU (second choice, line 25–26). Regardless if the second CPU is busy or not, FALCON uses it for balancing softirqs.

The fundamental challenge is the gap between fine-grained, distributed, per-packet balancing and the complexity of achieving global load balance. To overcome it, FALCON devises a dynamic softirq balancing algorithm that 1) prevents overloading any core and 2) maintains a reasonably good balance across cores 3) at a low cost. As shown in Algorithm 1, the dynamic balancing algorithm centers on two designs.

Algorithm 1 Dynamic Softirq Balancing

Variables: socket buffer *skb*; current average load of the system L_{avg}; network flow hash *skb.hash* and device ID *ifindex*; FALCON CPU set FALCON_CPUS.
// Stage transition function
function NETIF_RX(*skb*)
    // Enable FALCON *only if there is room for parallelization*
    if L_{avg} ¡ FALCON_LOAD_THRESHOLD **then**
        cpu := get_falcon_cpu(*skb*)
        // Enqueue skb to cpu's packet queue and raise softirq
        enqueue_to_backlog(*skb*, *cpu*)
    else
        // Original execution path (RPS or current CPU)
        ...
    end if
end function
// Determine where to place the next softirq
function GET_FALCON_CPU(*skb*)
    // First choice based on device hash
    hash := hash_32(*skb.hash* + *ifindex*)
    cpu := FALCON_CPUS[*hash* % NR_FALCON_CPUS]
    if *cpu.load* ¡ FALCON_LOAD_THRESHOLD **then**
        return *cpu*
    end if
    // Second choice if the first one is overloaded
    hash := hash_32(*hash*)
    return FALCON_CPUS[*hash* % NR_FALCON_CPUS]
end function

First, FALCON is enabled only when there are sufficient CPU resources to parallelize individual network flows otherwise all softirqs stay on the original core (line 6–9). FALCON monitors system-wide CPU utilization and switches softirq pipelining and splitting on and off according to `FALCON_LOAD_THRESHOLD`. Second, FALCON employs a *two-choice* algorithm for balancing softirqs: 1) it first computes a hash on the device ID and the flow key to uniquely select a CPU for processing a softirq (line 19–20). Given the nature of hashing, the first choice is essentially a uniformly random CPU in the FALCON CPU set. This helps evenly spread softirqs across CPUs without quantitatively comparing their loads. If the first selected CPU is busy, FALCON performs double hashing to pick up another CPU (second choice, line 25–26). Regardless if the second CPU is busy or not, FALCON uses it for balancing softirqs.

3.5 Implementation

We have implemented FALCON upon Linux network stack in two generations of Linux kernel, 4.19 and 5.4. Underpinning FALCON's implementation, there are two specific techniques:

3.5.1 Stage Transition Functions

To realize *softirq pipelining and splitting*, FALCON re-purposes a state transition function, `netif_rx` (line 4–14 of Algorithm 1), and explicitly inserts it at the end of each network device's processing path. Therefore, once a packet finishes its processing on one network device, it could be steered by `netif_rx` to a different CPU core for the subsequent processing. The `netif_rx` function relies on the CPU-selection function `get_falcon_cpu` (line 17–27) to choose a target CPU (line 7), enqueues the packet to the target CPU's packet processing queue (line 8), and raises a softirq to signal the target CPU (also line 8).

Furthermore, in the current implementation of *softirq splitting*, FALCON splits two heavy processing functions of the first network device (i.e., physical NIC) — `skb_allocation` and `napi_gro_receive` — onto two separate cores by inserting `netif_rx` right before the `napi_gro_receive` function. We call this approach "GRO-splitting". Note that, to apply such a splitting approach, we need to identify that the two split functions are "stateless" — the processing of one function does not depend on the other function.

3.5.2 Hashing-Based Load Balancing Mechanism

FALCON employs a two-choice *dynamic load balancing* algorithm (line 17–27), which relies on a new hashing mechanism to pick up the target CPU. Specifically, the first CPU choice is determined by the hash value (line 19) calculated upon both the flow information `skb.hash` and device information `ifindex` — `skb.hash` represents the *flow hash*, calculated only once when a packet enters the first network device and based on the flow key (`flow_keys`); `ifindex` represents the unique device index of a network device. With this hash value, FALCON ensures that 1) given the same flow but different network devices, hash values are distinct — a flow's multiple process stages of devices can be distinguished; 2) given the same network device, all packets of the same flow will always be processed on the same core — preserving processing dependencies and avoiding "out-of-order" delivery; 3) FALCON does not need to store the "core-to-device" mapping information; instead, such mapping information is captured by the hash value, inherently. Furthermore, if the first CPU choice fails (i.e., the selected CPU is busy), FALCON simply generates a new hash value for the second choice (line 25).

FALCON is enabled when the average system load (i.e., CPU usage) is lower than `FALCON_LOAD_THRESHOLD` (line 6); otherwise, it is disabled (line 11) indicating no

sufficient CPU resources for packet parallelization. FALCON maintains the average system load in a global variable L_{avg} and updates it every N timer interrupts within the global timer interrupt handler (i.e., `do_timer`), via reading the system state information (i.e., `/proc/stat`) to detect each core's load.

3.6 Evaluation

We evaluate both the effectiveness of FALCON in improving the performance of container overlay networks. Results with micro-benchmarks demonstrate that 1) FALCON improves throughput up to within 87% of the native performance in UDP stress tests with a single flow, 2) significantly improves latency for both UDP and TCP, and 3) achieves even higher than native throughput in multi-flow TCP tests. Experiments with two generations of Linux kernels that have undergone major changes in the network stack prove FALCON's effectiveness and generality. Results with real applications show similar performance benefits. Nevertheless, overhead analysis reveals that FALCON exploits fine-grained intra-flow parallelism at a cost of increased CPU usage due to queue operations and loss of locality, which in certain cases could diminish the performance gain.

3.6.1 Experimental Configurations

The experiments were performed on two DELL PowerEdge R640 servers equipped with dual 10-core Intel Xeon Silver 4114 processors (2.2 GHz) and 128 GB memory. Hyperthreading and turbo boost were enabled, and the CPU frequency was set to the maximum. The two machines were connected directly by two physical links: Intel X550T 10-Gigabit Ethernet (denoted as 10G), and Mellanox ConnectX-5 EN 100-Gigabit Ethernet (denoted as 100G). We used Ubuntu 18.04 with Linux kernel 4.19 and 5.4 as the host OSes. We used the Docker overlay network mode in Docker version

19.03.6 as the container overlay network. Docker overlay network uses Linux's builtin VXLAN to encapsulate container network packets. Network optimizations (e.g., TSO, GRO, GSO, RPS) and interrupt mitigation (e.g., adaptive interrupt coalescing) were enabled for all tests.

For comparisons, we evaluated the following three cases:

- *Native host*: running tests on the physical host network without containers (denoted as *Host*).
- *Vanilla overlay*: running tests on containers with default docker overlay network (denoted as *Con*).
- *Falcon overlay*: running tests on containers with Falcon-enabled overlay network (denoted as *Falcon*).

3.6.2 Micro-Benchmarks

Single-flow stress test. As shown in Figure 3.2, UDP workloads suffer higher performance degradation in overlay networks compared to TCP. Unlike TCP, which is a connection-oriented protocol that has congestion (traffic) control, UDP allows multiple clients to send packets to an open port, being able to press the network stack to its limit on handling a single flow. Since FALCON addresses softirq serialization, the UDP stress test evaluates its maximum potential in accelerating single flows. If not otherwise stated, we used 3 sockperf clients to overload a UDP server. Experiments were performed in Linux version 4.19 and 5.4. The new Linux kernel had major changes in `sk_buff` allocation, a data structure used throughout the network stack. Our study revealed that the new kernel achieves performance improvements as well as causing regressions.

Figure 3.11 shows that FALCON achieved significant throughput improvements over Docker overlay, especially with large packet sizes. It delivered near-native

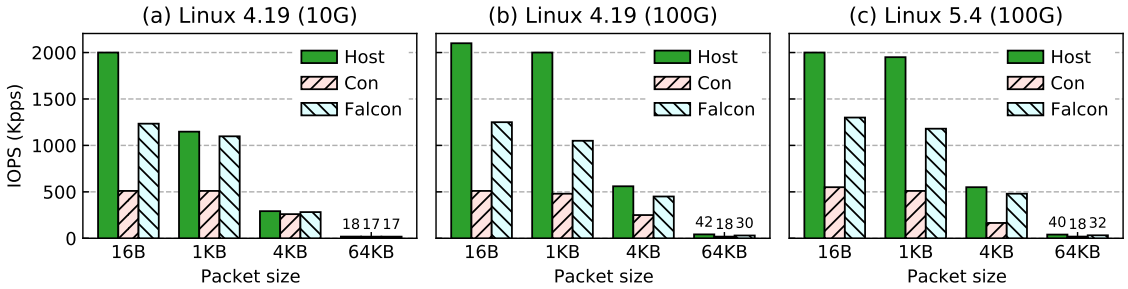

Figure 3.11: Packet rates in host network, vanilla overlay, and FALCON overlay under a UDP stress test.

throughput in the 10 Gbps Ethernet while bringing packet rate up to 87% of the host network in the 100 Gbps Ethernet. However, there still existed a considerable gap between FALCON and the host network for packets smaller than the maximum transmission unit (MTU) in Ethernet (i.e., 1500 bytes).

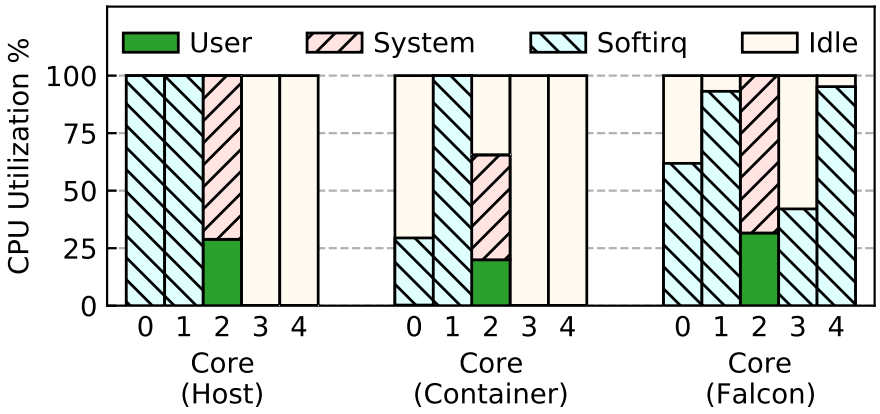

Figure 3.12: CPU utilization of a single UDP flow.

Figure 3.12 shows the breakdown of CPU usage on multiple cores for the 16B single-flow UDP test in the 100 Gbps network. With the help of packet steering, network processing in the vanilla Linux can utilize at most three cores – core-0 for hardirqs and the first softirq responsible for packet steering, core-1 for the rest of softirqs, and core-2 for copying received packets to user space and running application

threads. It can be clearly seen that core-1 in the vanilla overlay was overloaded by the prolonged data path with three softirqs. In comparison, FALCON is able to utilize two additional cores to process the two extra softirqs. The CPU usage also reveals that both the host network and FALCON were bottlenecked by user space packet receiving on core-2. Since FALCON involves packet processing on multiple cores, it is inevitably more expensive for applications to access packets due to loss of locality. This explains the remaining gap between FALCON and the host network. To further narrow the gap, the user space applications need to be parallelized, which we leave for future work.

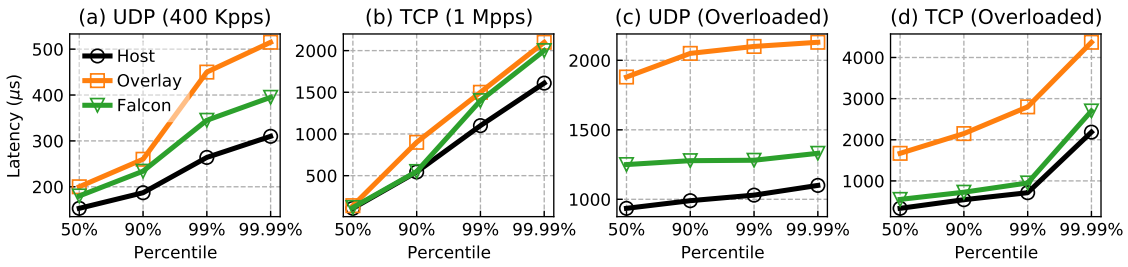

Figure 3.13: Effect of FALCON on per-packet latency. Packet size is 16 B in (a, c, d) and 4 KB in (b).

Single-flow latency. Figure 3.13 depicts per-packet latency in single-flow UDP and TCP tests. We are interested in latency in both 1) underloaded tests, wherein client sending rate is fixed in all three cases to avoid overloading any cores on the receiving side, and 2) overloaded tests, in which each case is driven to its respective maximum throughput before packet drop occurs. In the underloaded UDP test in Figure 3.13 (a), FALCON had modest improvements on the average and 90^{th} percentile latency and more pronounced improvements towards the tail. Note that fine-grained softirq splitting, such as GRO splitting, did not take effect in UDP since GRO was not the bottleneck. In contrast, Figure 3.13 (c) suggests that softirq pipelining helped

tremendously in the overloaded UDP test wherein packets processed on multiple cores experienced less queuing delay than that on a single core.

Figure 3.13 (b) and (d) shows the effect of FALCON on TCP latency. Our experiments found that in the overloaded TCP test (Figure 3.13 (d)), latency is largely dominated by queuing delays at each network device and hence the improvement is mainly due to softirq pipelining while softirq splitting may also have helped. It is worth noting that FALCON was able to achieve near-native latency across the spectrum of average and tail latency. For underloaded TCP test with packets less than 4 KB (not listed in the figures), neither softirq splitting nor pipelining had much effect on latency. For 4 KB underloaded TCP test (Figure 3.13 (b)), GRO splitting helped to attain near-native average and the 90^{th} percentile latency but failed to contained the long tail latency. We believe this is due to the possible delays in inter-processor interrupts needed for raising softirqs on multiple cores. It is worth noting that despite the gap from the host network FALCON consistently outperformed the vanilla overlay in all cases.

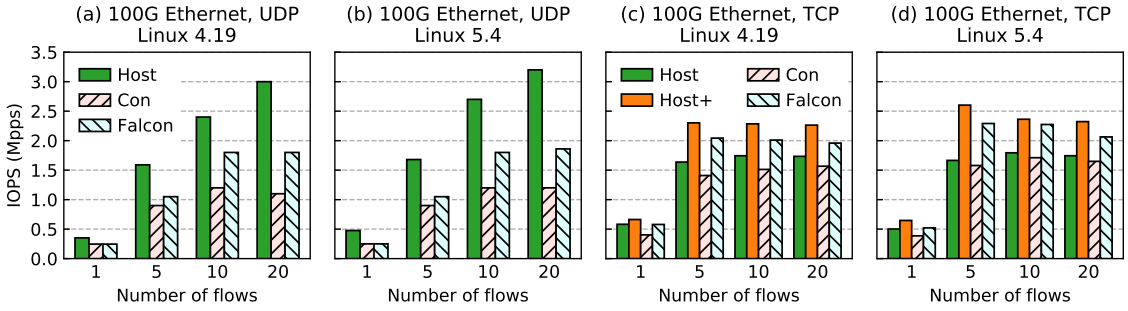

Figure 3.14: Packet rates in host network, vanilla overlay, and FALCON under multi-flow UDP and TCP tests.

Multi-flow throughput. This sections compares FALCON with existing packet steering techniques (i.e., RSS/RPS) in *multi-flow* tests — multiple flows were hosted

within *one* container. In all tests, both RSS and RPS were enabled and we used dedicated cores in `FALCON_CPUS`. This ensures that FALCON always has access to idle cores for flow parallelization. As previously discussed, GRO-splitting is only effective for TCP workloads and hence does not take effect in UDP tests. The packet sizes were set to 16 B and 4 KB for UDP and TCP, respectively. Unlike the UDP stress test, which used multiple clients to press a single flow, the multi-flow test used one client per flow. Figure 3.14 (a) and (b) show that FALCON can consistently outperform the vanilla overlay with packet steering by as much as 63%, within 58% to 75% of that in the host network. Note that FALCON neither improved nor degraded performance for a single flow. It is because, for UDP tests with 16 B packets without using multiple clients, the sender was the bottleneck.

For TCP multi-flow tests, we further enabled GRO-splitting for the host network (denoted as *Host+*). Figure 3.14 (c) and (d) show that GRO processing is a significant bottleneck even for the host network. GRO-splitting helped achieve up to 56% throughput improvement in *Host+* than that in the vanilla host network. With FALCON, the overlay network even outperformed *Host* by as much as 37%.

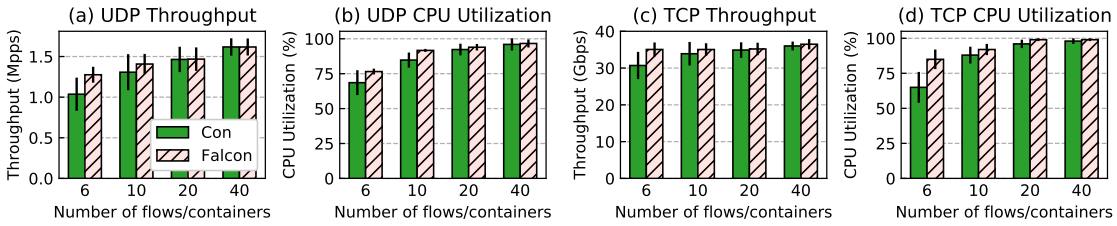

Figure 3.15: FALCON's benefit diminishes as utilization increases but causes no performance loss when system is overloaded.

Multi-container throughput in busy systems. This section evaluates FALCON in more realistic scenarios in which multiple containers, each hosting one flow, are

running in a busy system. Unlike the multi-flow tests that used dedicated, idle cores for flow parallelization, in the multi-container tests all cores were actively processing either hardirqs, softirqs, or application threads. FALCON needed to exploit idle CPU cycles on unsaturated cores for flow parallelization. This evaluates the effectiveness of the dynamic balancing algorithm. We limited the packet receiving CPUs to 6 cores [4] and configured them as `FALCON_CPUS`. As illustrated in Figure 3.15, we gradually increased the number of containers from 6 to 40 in order to drive the receiving cores from around 70% busy until overloaded. We observed that: 1) when the system had idle CPU cycles (e.g., under 6 or 10 containers), FALCON was able to improve overall throughput by up to 27% and 17% under UDP and TCP, respectively. In addition, FALCON's performance was more consistent across runs compared to the vanilla container network; 2) when the system was pressed towards fully saturated (e.g., 100% utilization with 20 and more containers), FALCON's gain diminished but never underperformed RSS/RPS. Figure 3.15 (b) and (d) show that FALCON's diminishing gain was observed during high CPU utilization and FALCON was disabled once system is overloaded.

Parameter sensitivity. FALCON is disabled when the system load is high since there is a lack of CPU cycles for parallelization. In this section, we evaluate the effect of parameter `FALCON_LOAD_THRESHOLD`, which specifies the utilization threshold for disabling FALCON. Figure 3.16 shows that always enabling FALCON (denoted as *always-on*) hurt performance when the system was highly loaded while setting a low utilization threshold (e.g., 70% and lower) missed the opportunities for parallelization. Our empirical studies suggested that a threshold between 80-90% resulted in the best performance.

[4] It was impractical for us to saturate a 40-core system due to limited client machines; hence we selected a subset of cores for evaluation.

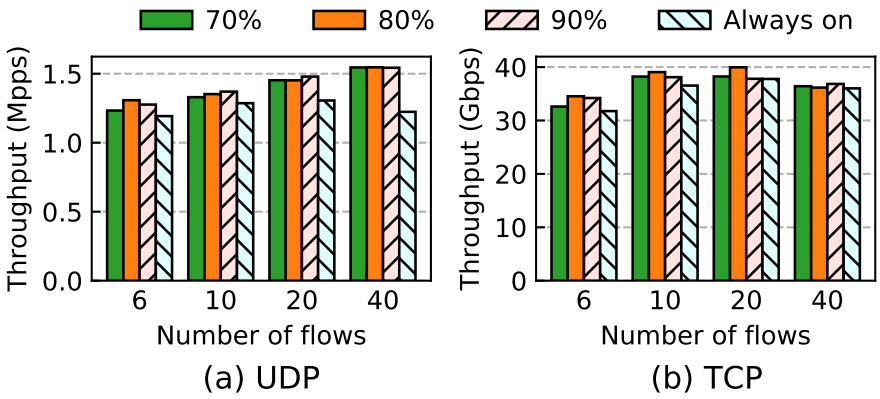

Figure 3.16: Effect of the average load threshold and its impact on container network performance.

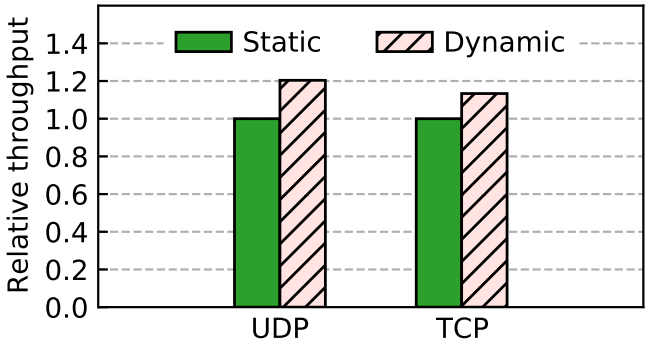

Figure 3.17: FALCON adapts to changing workload and re-balances softirqs dynamically.

Adaptability test. To demonstrate the significance of FALCON's two-random choice algorithm, we created hotspots by suddenly increasing the intensity of certain flows. In a hashing-based balancing algorithm, such as RSS/RPS and the first random choice in FALCON, the softirq-to-core mapping is fixed, thereby unable to adapt to workload dynamics. In contrast, FALCON's *two-choice* dynamic re-balancing algorithm allows some softirqs to be steered away from an overloaded core and quickly resolves the bottleneck. In the test, we randomly increased the intensity of one flow, resulting in one overloaded core. We compare FALCON's two-choice balancing algorithm (denoted as *dynamic*) with static hashing (FALCON' balancing algorithm with

the second choice disabled, denoted as *static*). As shown in Figure 3.17, the two-choice balancing algorithm achieved 18% higher throughput in UDP about 15% higher throughput in TCP, respectively. Most importantly, the performance benefit was consistent across multiple runs. These results suggest that the two-choice balancing algorithm can effectively resolve transient bottlenecks without causing fluctuations.

3.6.3 Application Results

Web serving. We measured the performance of the Cloudsuite's Web Serving benchmark [48] with FALCON. Cloudsuite Web Serving, which is a benchmark to evaluate page load throughput and access latency, consists of four tiers: an *nginx* web server, a *mysql* database, a *memcached* server and clients. The web server runs the Elgg [49] social network and connects to the cache and database servers. The clients send requests, including login, chat, update, etc., to the social network (i.e., the web server). We evaluated the performance with our local testbed. Web server's `pm.max_children` was set to 100. The cache and database servers were running on two separate cores to avoid interferences. All clients and servers ran inside containers and were connected through Docker overlay networks on top of the 100 Gbps NIC.

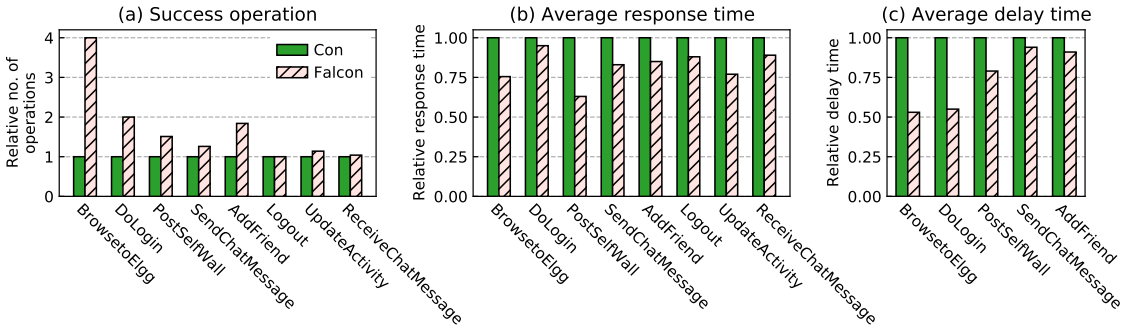

Figure 3.18: FALCON improves the performance of a web serving application in terms of higher operation rate and lower response time.

Figure 3.18(a) shows the "success operation" rate with a load of 200 users under the *vanilla* overlay network and FALCON. Compared to the *vanilla* case, FALCON improves the rate of individual operations significantly, by up to *300%* (e.g., `BrowsetoElgg`). Figure 3.18(b) and (c) illustrate the average response time and delay time of these operations: The response time refers to the time to handle one request, while the delay time is the difference between the target (expected time for completion) and actual processing time. With FALCON, both response time and delay time are significantly reduced. For instance, compared to the *vanilla* case, the maximum improvement in average response time and delay time is 63% (e.g., `PostSelfWall`) and 53% (e.g., `BrownsetoElgg`), respectively. FALCON's improvements on both throughput and latency are mainly due to distributing softirqs to separate cores, thus avoiding highly loaded cores.

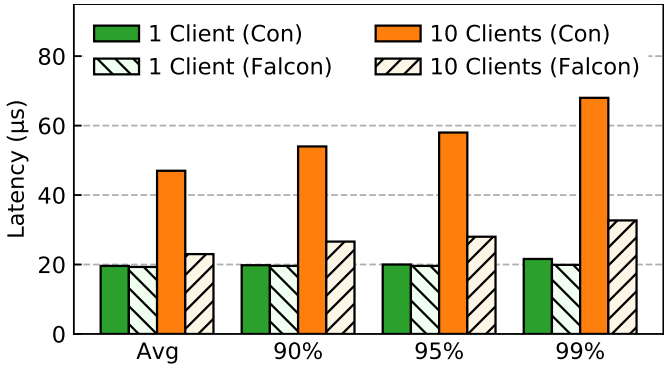

Figure 3.19: FALCON reduces the average and tail latency under data caching using Memcached.

Data caching. We further measured the average and tail latency using Cloudsuite's data caching benchmark, *memcached* [50]. The client and server were running in two containers connected with Docker overlay networks. The *memcached* server was configured with 4GB memory, 4 threads, and an object size of 550 bytes. The client

had up to 10 threads, submitting requests through 100 connections using the Twitter dataset. As shown in Figure 3.19, with one client, FALCON reduces the tail latency (99^{th} percentile latency) slightly by 7%, compared to the *vanilla* case. However, as the number of clients grows to ten, the average and tail latency (99^{th} percentile latency) are reduced much further under FALCON, by 51% and 53%. It is because, as the number of clients (and the request rate) increases, kernel spends more time in handling interrupts, and FALCON greatly increases its efficiency due to pipelined packet processing and balanced software interrupts distribution.

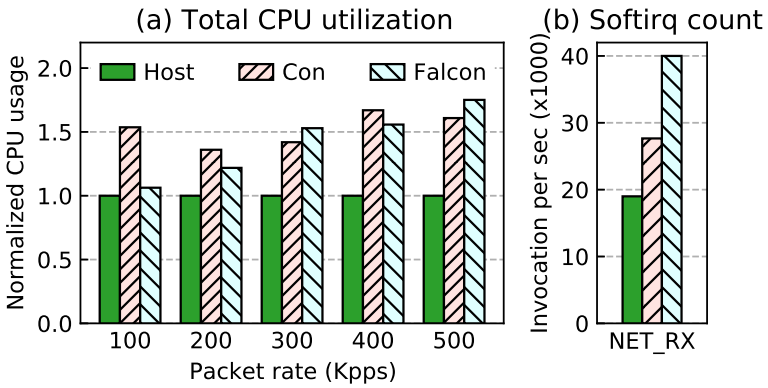

Figure 3.20: Overhead of FALCON.

3.6.4 **Overhead Analysis**

The overhead of FALCON mainly comes from two sources: interrupt redistribution and loss of packet data locality. These are inevitable, as FALCON splits one softirq into multiple ones to help packets migrate from one CPU core to another. Note that, the essence of FALCON is to split and spread CPU-consuming softirqs to multiple available CPUs instead of reducing softirqs. As the overhead ultimately results in higher CPU usage given the same traffic load, we quantify it by measuring

the total CPU usage with fixed packet rates. Figure 3.20 shows the CPU usage with a 16B single-flow UDP test under various fixed packet rates in three network modes: native host, vanilla overlay, and FALCON.

As depicted in Figure 3.20 (a), compared to vanilla overlay, FALCON consumes similar (or even lower) CPU resources when the packet rate is low, while slightly more CPU resources ($\leq$ 10%) when the packet rate is high. Meanwhile, FALCON triggers more softirqs, e.g., by 44.6% in Figure 3.20 (b).[5] It indicates that though FALCON could result in loss of cache locality as the processing of a packet is spread onto multiple cores, it brings *little* CPU overhead compared to the vanilla overlay. It is likely because the vanilla overlay approach does not have good locality either, as it needs to frequently switch between different softirq contexts (e.g., for NIC, VXLAN, and veth) on the same core. As expected, FALCON consumes more CPU resources compared to native host, and the gap widens as the packet rate increases.

3.6.5 Discussion

Dynamic softirq splitting. While we found softirq splitting is necessary for TCP workloads with large packets and can significantly improve both throughput and latency, it may impose overhead for UDP workloads that are not bottlenecked by GRO processing. In the meantime, we employ offline profiling to determine the functions within a softirq that should be split and require the kernel to be recompiled. Although FALCON can be turned on/off completely based on the system load, there is no way to selectively disable function-level splitting while keeping the rest part of FALCON running. As such, certain workloads may experience suboptimal performance under GRO splitting. One workaround is to configure the target CPU for softirq splitting

[5]Note that the overlay network triggers fewer softirqs in Figure 3.20 (b) than that in Figure 3.4, as we measured it in a less loaded case (400 Kpps).

to use the same core so that the split function is never moved. We are investigating a dynamic method for function-level splitting.

Real-world scenarios. FALCON is designed to be a general approach for all types of network traffic in container overlay networks. Particularly, two practical scenarios would greatly benefit from it: 1) Real-time applications based on "elephant" UDP flows, such as live HD streaming, VoIP, video conferencing, and online gaming; 2) a large number of flows with unbalanced traffic — multiple flows could co-locate on the same core if the number of flows is larger than the core count, where FALCON can parallelize and distribute them evenly. Note that, FALCON's effectiveness depends on access to idle CPU cycles for parallelization. In a multiple-user environment, policies on how to fairly allocate cycles for parallelizing each user's flows need to be further developed.

3.7 Conclusion

This chapter demonstrates that the performance loss in overlay networks due to serialization in the handling of excessive, expensive softirqs can be significant. We seek to parallelize softirq processing in a single network flow and present FALCON, a fast and balanced container network. FALCON centers on three designs: softirq pipelining, splitting, and dynamic balancing to enable fine-grained, low-cost flow parallelization on multicore machines. Our experimental results show that FALCON can significantly improve the performance of container overlay networks with both micro-benchmarks and real-world applications.

CHAPTER 4

ACCELERATING PACKET PROCESSING IN CONTAINER OVERLAY NETWORKS VIA PACKET-LEVEL PARALLELISM

Overlay networks serve as the *de facto* network virtualization technique for providing connectivity among distributed containers. Despite the flexibility in building customized private container networks, overlay networks incur significant performance loss compared to physical networks (*i.e.*, the native). The culprit lies in the inclusion of multiple network processing stages in overlay networks, which prolongs the network processing path and overloads CPU cores. In this chapter, we propose **mFlow**, a novel packet steering approach to parallelize the in-kernel data path of network flows. mFlow exploits *packet-level* parallelism in the kernel network stack by splitting the packets of the same flow into multiple micro-flows, which can be processed in parallel on multiple cores. mFlow devises new, generic mechanisms for flow splitting while preserving in-order packet delivery with little overhead. Our evaluation with both micro-benchmarks and real-world applications demonstrates the effectiveness of mFlow, with significantly improved performance – *e.g.*, by 81% in TCP throughput and 139% in UDP compared to vanilla overlay networks. mFlow even achieved higher TCP throughput than the native (*e.g.*, 29.8 vs. 26.6 Gbps).

4.1 Introduction

Due to high portability, high density, low performance overhead, and low operational cost, containers have quickly gained popularity and become adopted by high performance computing systems (HPC) [51, 52, 53, 54, 55, 56, 57, 58, 59]. Unlike

VMs, containers achieve *lightweight* virtualization by running directly on the host operating systems (OS) – *i.e.*, *no* guest OSes and virtual hardware emulation involved – while isolation between containers remains enforced through kernel-level features such as namespaces [60], cgroups [61], and seccomp [62].

However, containers are *no longer lightweight* with regard to peripheral components, especially for networking. Recent studies [2, 1, 14] revealed that compared to the native (*i.e.*, no virtualization), containers achieved ∼50% less network throughput and suffered much higher packet-level processing latency. The culprit of the poor container network performance lies in the complexity of constructing network connections: Containers rely on *overlay networks* – the de facto network virtualization technique in containers – allowing each container to have its own network namespace and private IP address while being independent of the host network. The construction of overlay networks requires a set of software network devices, such as `VxLAN` [12] for packet encapsulation/decapsulation, `veth` for virtual network interfaces of containers, and virtual bridges (*e.g.*, Linux bridge or Open vSwitch [13]) to connect them. The involvement of multiple software network devices prolongs the data path of container network packets, inevitably incurring additional overhead and delays to packet processing with high CPU usage [2, 14].

Worse, since the Linux kernel typically squeezes all the processing stages of a single flow on a single CPU core [2], the computation of packet processing can easily overload the core, thus throttling the network throughput of the flow. This negatively impacts the performance and scalability of many HPC workloads, such as live HD streaming, distributed machine learning tasks, and big data processing tasks – typically generating *long-lived, high-throughput* flows, known as "elephant" flows. For example, due to such a CPU bottleneck, distributed machine learning tasks stopped scaling after only consuming 25 Gbps out of a 100 Gbps network link [63].

This chapter investigates how and to which degree in-kernel packet processing can be optimized to accelerate container overlay networks. Ideally, the above-mentioned CPU bottleneck can be addressed/mitigated if we can effectively convert any elephant flow into multiple mouse flows, each containing a small portion of the flow's packets and being processed upon a separate core. Several instant benefits are: (1) Each mouse flow contains fewer packets, thus avoiding overloading a single core (even for a heavyweight network device); (2) Packets of different mouse flows can be processed in parallel, thus accelerating packet processing speed; (3) It can more efficiently leverage a multi-core system to mix and balance elephant and mouse flows – *i.e.*, an elephant flow is just equivalent to a bunch of mice flows.

To seek the feasibility of this idea, we design and develop **mFlow** – a novel approach to parallelize *in-kernel* data path of (elephant) flows. mFlow exploits fine-grained, *packet-level* parallelism based on an often overlooked fact: While existing in-kernel packet processing requires all packets of a single flow to be processed in a pipelined manner (in sequence), in-order packet processing does *not* need to be strictly guaranteed at all times along the *stateless* network path, but instead only when necessary (for the *stateful* path), *e.g.*, before packets enter the transport layer (*i.e.*, TCP) or are sent to user-space applications. Upon this observation, mFlow achieves packet-level parallelism by splitting the packets of the same flow into multiple small batches, called *micro-flows*, which can be processed *in parallel* on multiple cores. mFlow devises generic packet steering mechanisms for *in-kernel flow splitting* that can be enabled at any point of the stateless network path.

One key challenge to mFlow lies in that as each CPU core may have different processing capability and/or be interrupted by concurrent kernel tasks, packets of different micro-flows may not preserve their arrival order after parallel processing – *out-of-order* packet delivery causes incorrectness (in TCP) or poor user experiences (in

UDP). This is precisely why the existing in-kernel network stack processes packets in order, thus only needing to reorder a small number of packets that are delayed during transmission. Although MFLOW can leverage the kernel's packet reordering mechanism to ensure all packets are still in order after parallel processing, the packet-level reordering incurs significant overhead. MFLOW addresses this issue in two ways: (1) by choosing a suitable batch size for micro-flows, the number of out-of-order packets can be dramatically reduced; (2) instead of reordering packets at a per-packet level, MFLOW devises a batch-based flow reassembling mechanism incurring *little* overhead.

We know of no other kernel techniques supporting packet-level parallelism for accelerating container overlay networks. We have implemented a prototype of MFLOW in the Linux network stack (with kernel version 5.7). To summarize, in this chapter, we have made the following contributions:

- We perform a detailed investigation of the performance of container overlay networks and identify the main performance bottleneck for elephant flows to be the lack of sufficient network processing parallelism.
- We design and implement MFLOW, which explores packet-level packet processing parallelism in commodity OS kernel for fast overlay networks. Unlike existing approaches that only parallelize packet processing at a coarse-grained flow/device level, MFLOW allows a flow to be parallelized at any stateless stage along the network processing pipeline.
- Our evaluation of MFLOW using both micro-benchmarks and real-world applications shows that MFLOW can significantly improve network throughput (*e.g.*, by 81% in TCP and 139% in UDP compared to the vanilla overlay networks) and application-level performance (*e.g.*, by up to 7.5x for web serving). MFLOW

even achieves higher TCP throughput under container overlay networks than the native (*e.g.*, 29.8 vs. 26.6 Gbps) due to packet-level processing parallelism.

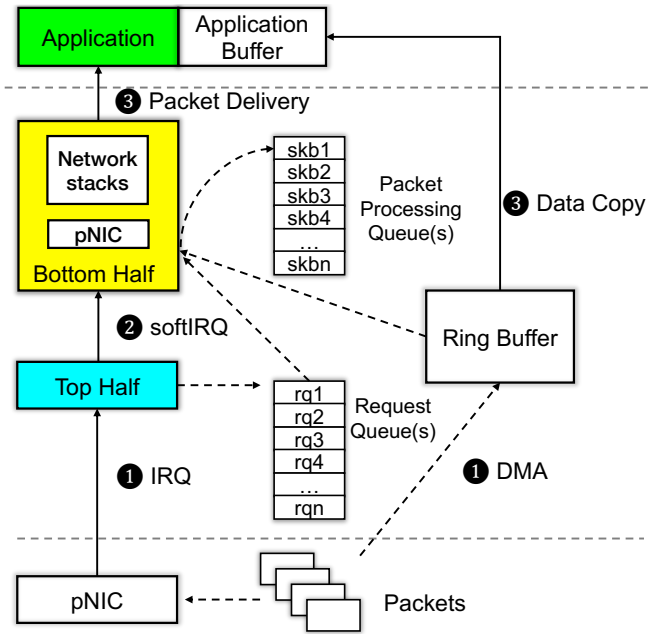

Figure 4.1: In-kernel packet processing.

4.2 Background and Motivation

4.2.1 Background

Packet processing: In-kernel packet processing, as illustrated in Figure 4.1, involves a complicated pipeline that traverses the physical network interface controller (`pNIC`), the kernel space, and the user space. We use packet reception as an example to demonstrate the process: When a packet arrives at the `pNIC`, in step ❶, it is copied (via DMA) to the kernel *ring buffer*, and the `pNIC` triggers a hardware interrupt (IRQ). The kernel is then invoked by the IRQ and starts the packet receiving process. The in-kernel receiving procedure further involves two parts: the top half and the bottom half.

The top half runs in the context of the IRQ, which simply marks that there is an incoming packet (in request queues) waiting for processing and notifies the bottom half (*i.e.*, by raising a software interrupt). The bottom half is then executed in the form of a software interrupt (softirq) (in step ❷). It serves as the main kernel network packet processing routine to process the packet through a set of network devices (*e.g.*, both physical and software NICs) and network protocol layers (*e.g.*, from layer 2 to layer 3/4). The Linux kernel uses a key data structure, `skb` (i.e., socket buffer), to represent each packet that can be freely manipulated and transferred across these network devices and layers. After a packet traverses all needed network devices and protocol layers along its path, it is finally delivered to the user-space application (in step ❸) — *i.e.*, the packet data/payloads (stored in the kernel ring buffer) is copied from the kernel buffer to the user-space application's buffer.

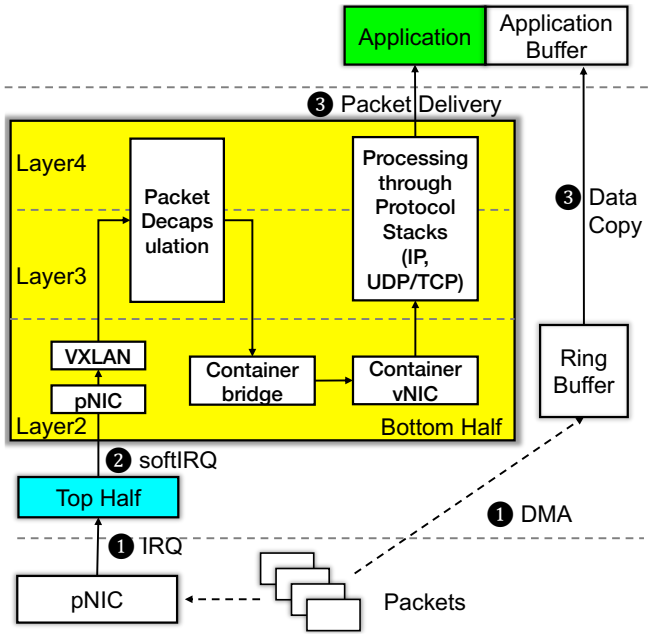

Figure 4.2: Container overlay networks.

Container overlay networks: Container overlay networks hinge on a tunneling technique (*e.g.*, through `VxLAN` [12]): When a container sends a packet (with private IPs), the overlay network encapsulates the packet in a new packet with the (source and destination) host IPs as the new packet header and the original packet as payload. When a container receives a packet, the overlay network decapsulates the received packet to recover the original packet and delivers it to the target containerized application using its private IP address.

As illustrated in Figure 4.2, the Linux kernel constructs the container overlay network with the help of several in-kernel software network devices – *i.e.*, a `VxLAN` network device for packet encapsulation/decapsulation, a virtual Ethernet device (`veth`) for virtual network interfaces of containers, and a virtual bridge (*e.g.*, Linux bridge or Open vSwitch [13]) to connect them. Hence, before a container packet is received by the user-space application, it needs to traverse *three* software devices and goes through the network protocol stacks *twice* — one for packet decapsulation and one for sending the decapsulated packet (by `veth`) to the user-space application. Throughout the whole process, one IRQ and three softirqs — *i.e.*, by `pNIC`, `VxLAN`, and `veth` — are raised. Therefore, compared to the native, the overlay network incurs prolonged data path with extra processing overhead.

Parallel packet processing: The prolonged data path in container overlay networks slows down per-packet processing and consumes more CPU cycles. By default, as the vanilla case shows in Figure 4.3, the Linux kernel squeezes all stages of a single flow's packet processing onto a single CPU core [1]. It is because the Linux network stack has been developed over the years and originally targeted less-powerful network

[1]The kernel thread for *packet delivery* – *i.e.*, copying data from the kernel ring buffer to the user-space buffer – is bonded with the core where the application thread runs; it can run on a separate core other than the in-kernel packet processing core(s).

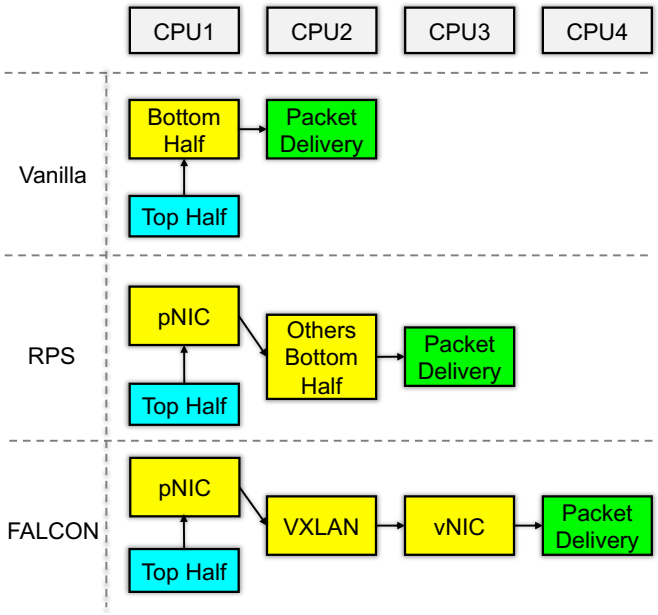

Figure 4.3: Parallel packet processing.

devices (*e.g.*, 1/10 Gbps) where a single core was powerful enough to handle a single network flow. However, in the face of today's high-performance, high-throughput network devices (*e.g.*, 100/400 Gbps), the CPU becomes the bottleneck – *i.e.*, packet processing can easily saturate a single core, preventing a single flow from achieving higher network throughput.

To leverage a multi-core system, both hardware and software *packet steering* approaches have been proposed to parallelize packet processing:

(1) Modern physical NICs enable multiple queues and apply receive side scaling (RSS) [26] to map different flows to separate cores (via hash values). This achieves *inter-flow* parallelism as different flows are associated with distinct hash values and can be mapped to different cores. Note that, it is common that one server can have more flows than available CPU cores; multiple flows might still be mapped to the same CPU core. The hardware-based parallelism mechanism, however, does *not* parallelize

a single (elephant) flow, as all packets from the same flow are assigned with the same hash value and hence processed on the same core.

(2) Receive packet steering (RPS) [27] in the Linux kernel is a software implementation of RSS, which realizes packet steering in the context of the first softirq (raised by `pNIC`'s IRQs) and again achieves *inter-flow* parallelism – *i.e.*, each flow is identified using a distinct hash value and mapped to a separate core. As the "RPS" case shows in Figure 4.3, for a single flow, RPS only separates the "top half" (as well as the first softirq) and the remaining "bottom half" onto two cores.

(3) Recent effort, FALCON [2], observed the lack of single-flow parallelization and enabled *device-level* and *function-level* parallelization for a single flow. As the "FALCON" case shows in Figure 4.3, packet processing stages associated with distinct network devices (`pNIC`, `VxLAN`, `vNIC`, etc.) can be distinguished and placed on separate cores by FALCON. However, one limitation of FALCON lies in that if a network device is heavy (*e.g.*, `VxLAN`), it can still saturate one CPU core and becomes the bottleneck. Further, the processing of a network packet in FALCON spans across multiple CPU cores, resulting in reduced data locality and extra queuing delays. Last, function-level parallelization in FALCON seems hard-coded and requires in-depth kernel code analysis.

4.2.2 Motivation

Experimental settings: To quantitatively analyze the effectiveness of existing parallel packet processing approaches, we evaluated the throughput and CPU utilization of the `VxLAN`-based overlay network using `sockperf` [41] (*i.e.*, a TCP/UDP traffic generator) between a pair of client and server machines. The machines were connected with Mellanox ConnectX-5 EN 100-Gigabit Ethernet adapters. Both the client and server had sufficient CPU and memory resources.

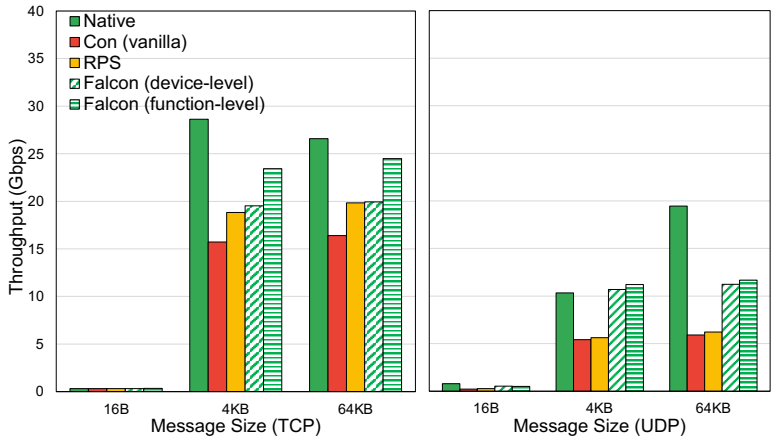

Figure 4.4: Throughput comparison under TCP/UDP with varying message sizes.

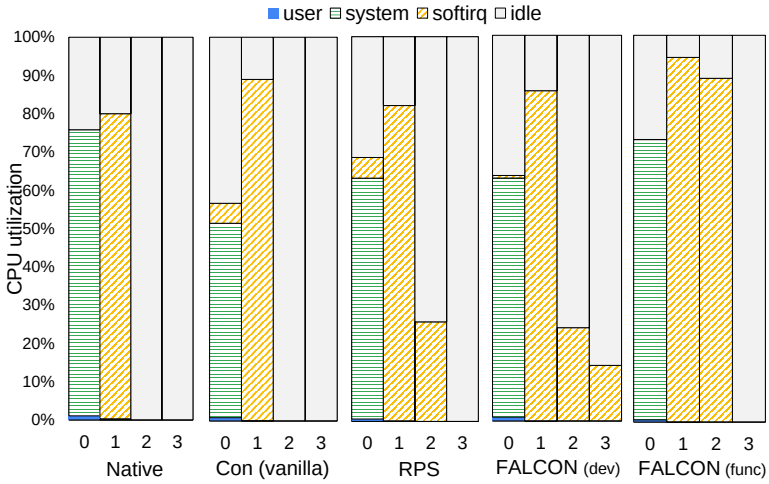

Figure 4.5: CPU utilization on separate cores (TCP with 64KB packet size).

Performance analysis: Figure 4.4 and Figure 4.5 depict the performance and CPU utilization comparisons between the native (*i.e.*, no containers), `VxLAN`-based container overlay network, RPS [27], and FALCON [2] using a single flow. We enabled the Linux kernel's default RPS mechanism. We downloaded FALCON's source code from its Github repository [64] and deployed its two parallelization approaches – at the device or function level.

Compared to the native, container overlay networks incurred higher performance overhead with significant performance drops – 40% for TCP and 80% for UDP under large message sizes (*e.g.*, 64 KB). The main reason is that: (1) container overlay networks entail prolonged data path with more software network devices as shown in Figure 4.2; (2) the Linux kernel by default places all packet processing of these devices on a single core, which easily overloads the core as indicated in Figure 4.5 (the container vanilla case) – softirqs of all network devices overloaded *core one* (close to 100%). Note that, Figure 4.5 shows average CPU utilization (*e.g.*, over 30 seconds). Although the average CPU% is under 100%, instant peak CPU% could reach 100% and throttle the performance, preventing a single flow from achieving higher throughput.

Compared to the vanilla overlay case, RPS slightly improved the throughput of container overlay networks – by 6% for UDP and 24% for TCP under large message sizes (*e.g.*, 64 KB). It is because, as shown in Figure 4.5 (the RPS case), RPS steered part of the softirqs from *core one* to *core two*, making *core one* capable of serving more packets. However, *core one* remained the bottleneck with high CPU usage, as the heavyweight network device – `VxLAN` (*i.e.*, part of the first softirq) – were still processed on core *one*.

To mitigate this, FALCON [2] distinguished different network devices and dispatched them onto separate cores, namely the *device-level* pipelining. As the example in Figure 4.5 shows, FALCON dispatched `VxLAN` to *core two* and placed the remaining devices on *core three*. In this way, FALCON increased the UDP throughput of container overlay networks significantly — by 80% (compared to vanilla overlay). However, it was still far below the native (only within 30%), because the device-level pipelining is still coarse-grained — *i.e.*, a heavy device/function can still saturate a single core.

Worse, the device-level pipelining merely worked for TCP with similar performance as RPS (in Figure 4.4). The reason is that, under TCP, heavyweight functions – *e.g.*, per-packet `skb` allocation and generic receive offload (`GRO`) [2] – remained on *core one* and overloading it, as depicted in Figure 4.5 (*i.e.*, the FALCON-dev case). To overcome this, the *function-level* pipelining in FALCON can further separate these functions onto separate cores. For example, by dispatching the `GRO` function (and all the following softirqs) on *core two*, FALCON increased the throughput of TCP – by 20% (compared to RPS). Meanwhile, *core one* again was overloaded – now purely by the `skb` allocation function (*i.e.*, the FALCON-func case in Figure 4.5) that cannot be parallelized by FALCON or any existing approaches.

Summary: Overlay networks incur non-trivial performance overhead for both TCP and UDP. State-of-the-art approaches can parallelize packet processing to a certain degree but encounter new bottlenecks. Hence, the performance of container overlay networks remains significantly lower than the native.

4.3 Design

To exploit in-kernel packet processing parallelism, we design and develop **mFlow** with the key ideas as follows: Instead of following the long-established pipelined, in-order processing, MFLOW exploits *packet-level* parallelism by splitting packets of the same flow into multiple small batches, called *micro-flows*, each being able to be processed on a separate core, called *splitting cores*. By doing this, multiple micro-flows of the same flow can be processed *in parallel* along the stateless network path and only reassembled before entering the stateful processing stage or user-space applications. As depicted in Figure 4.6, MFLOW can scale a heavyweight network device or even

[2] `GRO` reassembles small packets into larger ones to reduce per-packet processing overhead. We observed that the Linux kernel's `GRO` is mainly effective for TCP connections but not for UDP.

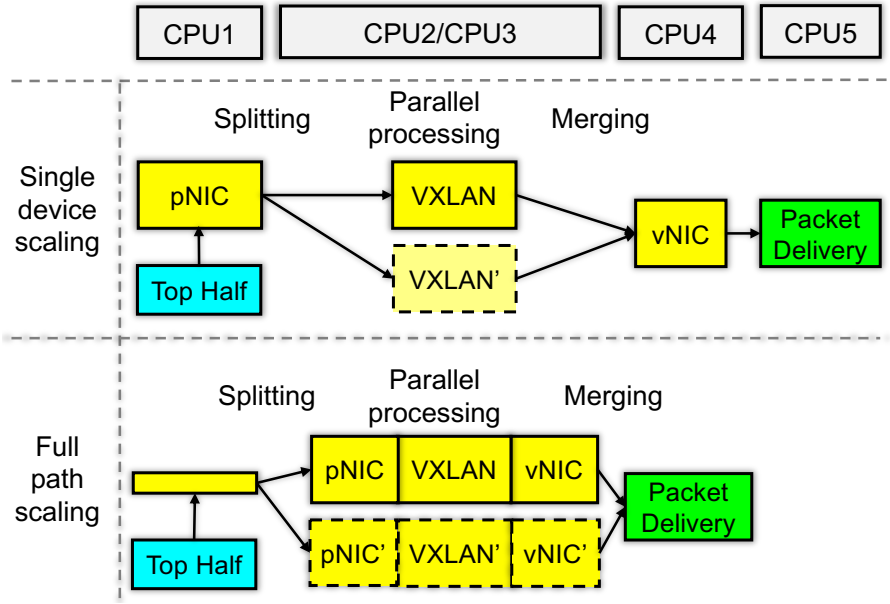

Figure 4.6: MFLOW achieves single device scaling or full path scaling via exploiting packet-level parallelism.

the full network path for a single flow. In the following sections, we present MFLOW's splitting mechanisms and how MFLOW efficiently preserves in-order packet delivery.

4.3.1 Flow Splitting

MFLOW does not re-design existing well-tested, mature kernel network stack, but instead realizes novel packet steering mechanisms to exploit packet-level parallelism. MFLOW devises two generic *mechanisms* for in-kernel flow splitting – *i.e.*, depending on whether the per-packet `skb` data structure is created or not. These splitting mechanisms enable MFLOW to either split a flow at a very early stage (*i.e.*, right after the first IRQ) or at any point along the stateless network processing path (*i.e.*, layer 2/3 and UDP layer).

Splitting mechanism along stateless network path: MFLOW splits a single flow by leveraging in-kernel *stage transition functions*. Specifically, during packet processing, a network packet – represented in the form of a *skb* data structure –

is transferred from one processing stage (*i.e.*, a network device) to another via a stage transition function (*e.g.*, `netif_rx`). The stage transition function enqueues the packet (*i.e.*, `skb`) into the queue of the device to be processed next on the same core. In this way, stage transition functions multiplex multiple stages of the flow in a pipelined manner on the same core – *i.e.*, once scheduled, each stage can process a batch of packets; stages are processed in an interleaved manner.

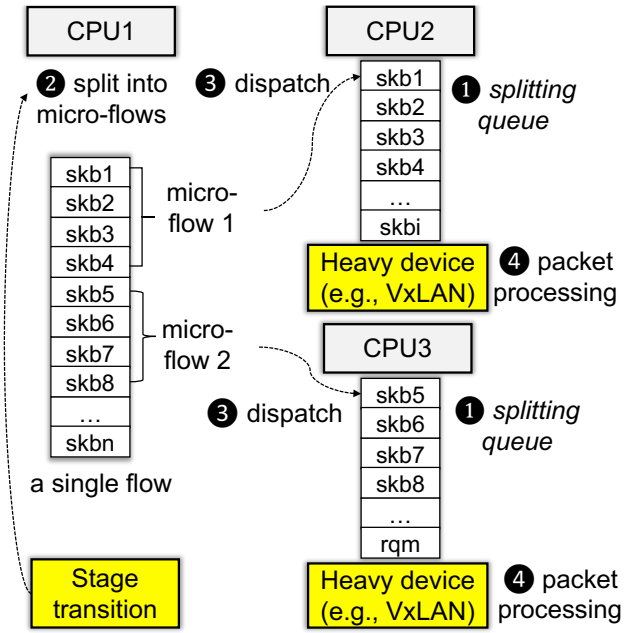

Figure 4.7: Flow-splitting function.

mFlow re-purposes the stage transition functions into a *flow-splitting function* for heavyweight network devices (in Figure 4.7): During network device initialization, for any network device (*e.g.*, `VxLAN`) that needs the packet-level parallelism, mFlow creates per-core, per-device *splitting queues* (❶). During packet processing, before any identified (elephant) flow enters the heavyweight network device, mFlow divides the packets of the flow into multiple small batches (❷). Each batch is called a *micro-flow* and covers a portion of the consecutive packets in the original flow. Then, mFlow

can select a distinct *splitting core* for a micro-flow and enqueues the packets of the micro-flow into its target core's splitting queue (❸). Meanwhile, a softirq is raised on the target splitting core via inter-processor interrupt (IPI). In this way, the bottom half of the network device will be executed later on all the involved splitting cores in parallel (❹).

This *flow-splitting function* works upon the per-packet `skb` data structure and can parallelize the processing of any stateless heavyweight network devices (or functions, *e.g.*, `GRO`). However, similar to the "FALCON-func" case in Figure 4.5, after MFLOW scales the heavyweight `VxLAN` device in container overlay networks via the flow-splitting function, we observed that the construction of the `skb` data structure (in the first stage of packet processing) became a heavy process – *i.e.*, it overloaded a single core.

To scale these heavyweight functions, we need a flow splitting mechanism that works at the earliest point of the network stack:

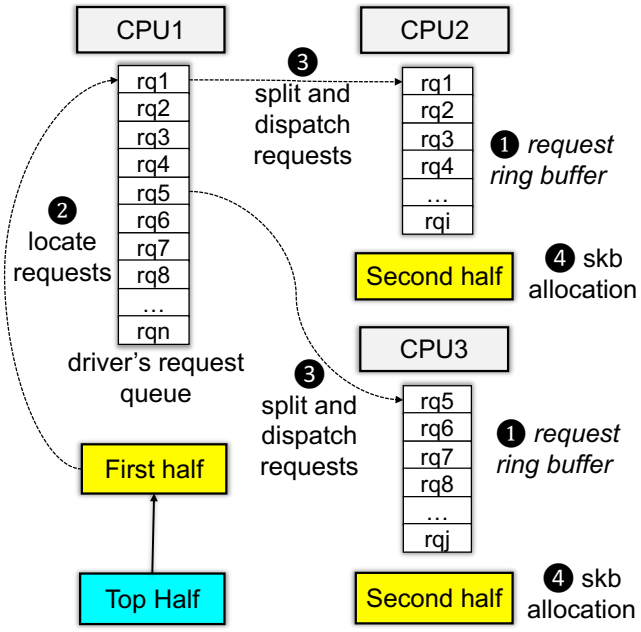

Figure 4.8: IRQ-splitting function.

Splitting mechanism for the first stage: Splitting the packets of a flow before `skb` allocation is challenging due to two factors: (1) It requires the support of the physical network device driver to locate raw packets. (2) As there is *no* `skb`, it needs a lightweight data structure to represent each raw packet, thus being able to dispatch them onto separate cores. To overcome these, mFlow devises an *IRQ-splitting function* to split/parallelize packet processing at the first stage:

As depicted in Figure 4.8, during the initialization of a flow that needs first stage parallelization, mFlow creates per-core *request ring buffers* on the splitting cores that will parallelize the first stage processing (❶). Then, the IRQ-splitting function divides the first stage – *i.e.*, the softirq context of the `pNIC` – into two halves. The first half (1) locates the incoming packet *requests* from the driver's request queue (❷) – *e.g.*, each request represents an incoming packet and contains information for the `skb` creation; (2) dispatches the requests onto target cores (❸) – similar to the above micro-flow based dispatching [3]; and (3) raises softirqs on target splitting cores (via IPIs). Finally, the second half will be invoked on the splitting cores to finish the remaining part of the original first stage – *e.g.*, `skb` allocation (❹). With this, mFlow can split and scale heavyweight functions at the earliest network software point by taking advantage of multiple cores. Note that, the design of the IRQ-splitting function relies *little* on a specific network device driver – *i.e.*, it only needs to know the driver's request queue and the way to locate its requests – making it portable to different network devices.

Parameters for packet-level parallelism: The degree of packet-level parallelism in mFlow is mainly determined by: (1) the number of outstanding packets; (2)

[3]Note that the IRQ-splitting function dispatches packet requests rather than `skb`s; it relies on the data structure of packet requests, created by device drivers, to represent each raw packet, hence being lightweight.

the batch size of micro-flows; and (3) the number of splitting cores. We discuss the implications of each parameter as follows:

For both TCP and UDP workloads, a number of *outstanding* packets could arrive at the receiver side approximately at the same time, especially for elephant flows. For example, given a TCP connection under the throughput of $\sim$ 30 Gbps, the sender (*e.g.*, iperf3 [31]) can issue $\sim$2,000 outstanding packets (with the size of MTU being 1,500 bytes) without receiving an ACK from the receiver. As there is no acknowledgment mechanism in UDP, a sender theoretically can issue as many outstanding packets as possible to the receiver.[4] As the outstanding packets arrive at the receiver approximately at the same time, dispatching them onto multiple cores enables packet-level parallelism. Therefore, the "heavier" a flow is, the more outstanding packets it produces and the higher the packet-level parallelism degree can be exploited.

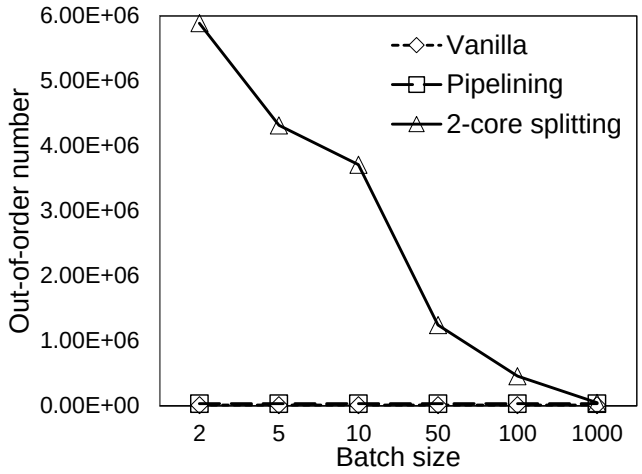

Figure 4.9: Number of out-of-order packet delivery vs. batch size of micro-flows (TCP with 64KB packet size).

[4]Practical UDP workloads implement congestion control upon the UDP protocol, which adjusts sending rate based on the observed quality of service such as packet loss, delay, jitter, etc.

Simply dispatching the outstanding packets of the same flow onto multiple cores may cause out-of-order delivery as different cores may not have a uniform processing speed. Though MFLOW's flow reassembling mechanism eventually preserves packet orders, more out-of-order delivery means additional effort for order preservation.

We observed that, in Figure 4.9, the number of out-of-order delivery after splitting reduced significantly as the batch size of micro-flows increased. When the batch size was set to 256 or above, *little* overhead was incurred for packet order preservation in MFLOW. Having a large batch size also preserves optimizations in packet processing. For example, GRO reassembles small packets into larger ones, thus reducing the number of packets to be processed. GRO can merge more consecutive small packets given a larger batch size. Batch size also has implications on load distribution: If all micro-flows have the same batch size and MFLOW evenly distributes them on multiple splitting cores, CPU utilization of each core would be similar (as packets go through similar processing).

Ideally, MFLOW can leverage as many cores as possible to exploit packet-level parallelism. However, in practice, the performance benefit may diminish as the core number increases due to multiple factors, such as the number of outstanding packets, batch size, queuing delay, and reassembling overhead. Our evaluation shows that using two cores for parallel packet processing greatly accelerates container overlay networks performance – *e.g.*, even higher than the vanilla native case. Further, as the original packet processing bottleneck has been mitigated by MFLOW, a new bottleneck arises due to data copying from the kernel to the user-space application.

4.3.2 **Flow Reassembling**

A key design goal of MFLOW is *not* to involve out-of-order packet delivery due to MFLOW's splitting mechanisms and parallel processing. We note that splitting a

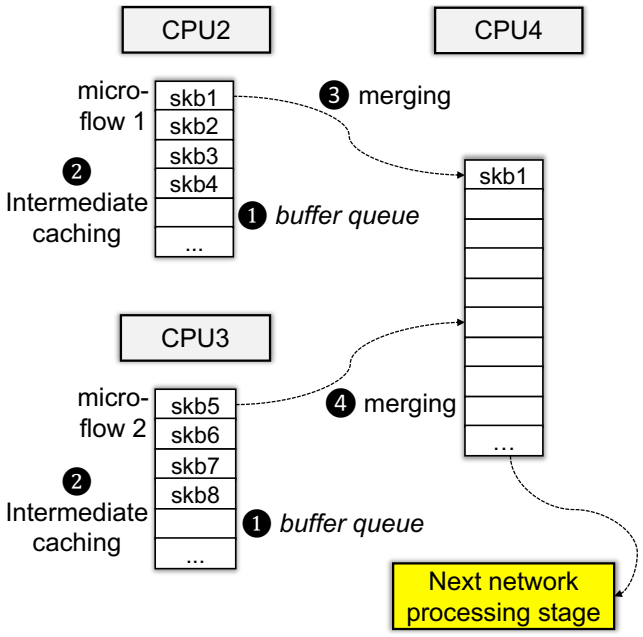

Figure 4.10: In-order flow reassembling.

single flow into micro-flows ensures that packets in each micro-flow naturally preserve their arrival orders. However, since each core may have different processing capability and/or be interrupted by other concurrent kernel tasks, packets of different micro-flows may not preserve their arrival orders after parallel processing.

To preserve the original sequences of micro-flows, mFlow devises an efficient *batch-based* flow reassembling mechanism. As depicted in Figure 4.10, for heavyweight network devices (or functions) that need packet-level parallelism, mFlow creates per-core, per-device *buffer queues* (❶). Then, for each splitting core that finishes the processing of a packet, it enqueues the packet to its buffer queue (❷), instead of directly sending it to the next processing stage. Meanwhile, each micro-flow is associated with an identifier which is incremented based on the position of the micro-flow in the original flow [5]. In other words, the ID reflects each micro-flow's order in the original flow. mFlow uses a global *merging counter* to keep track of the ID of the

[5] mFlow stores the ID information in each packet's `skb` data structure.

micro-flow being merged. To merge a micro-flow, mFlow (1) locates the buffer queue that stores the packets having the ID same as the *merging counter*; (2) fetches the packets from the buffer queue; and (3) sends them to the next processing queue/stage (❸ and ❹). mFlow keeps consuming packets from the same buffer queue until the next packet stores a different ID than the *merging counter*, indicating that mFlow should move to consume the next micro-flow. After mFlow increments the *merging counter*, it repeats step (1).

mFlow's batch-based flow reassembling approach has the following advantages: (1) The per-core, per-device buffer queues (used to cache intermediate micro-flows) ensure that each core can keep processing packets without being blocked by the merging process. (2) Packets are "re-ordered" on a per-batch basis, which is extremely efficient, especially compared to the kernel's existing per-packet reordering mechanism using an out-of-order queue. It also indicates that using a large batch size can significantly reduce merging overhead – *i.e.*, mFlow does not need to frequently switch between buffer queues to locate the next micro-flow.

Note that, although it makes intuitive sense to merge micro-flows right after a heavy device/function and before the next processing stage, we find that micro-flows can actually be merged as late as possible as long as the following packet processing is stateless (*i.e.*, no inter-packet processing dependency). For example, for UDP flows, micro-flows can be merged right before being delivered to user-space applications. The advantages for the late merging are as follows: (1) mFlow can reuse existing in-kernel backlog queues [6] as buffer queues with reduced queuing delay. (2) mFlow can parallelize the full packet processing path with fewer splitting cores (in Figure 4.6). (3) Packets are being processed on the same core with good data locality.

[6] In delivering packets to a user application, the kernel uses a backlog queue to store packets temporarily while the receive queue is being used by the application's receiving thread.

4.4 Implementation

We have implemented mFlow on the Linux network stack with kernel version 5.7 (∼600 LoC of addition or modification) with the focus on the presented splitting and reassembling mechanisms.

4.4.1 Flow-Splitting Function

mFlow implements the flow-splitting function by re-purposing a state transition function, `netif_rx`. Originally, such a state transition function enqueues a packet (*i.e.*, its `skb`) into the current core's *backlog queue* for future processing on the *same* core. mFlow, instead, splits received packets of a flow – that requires packet-level parallelism for a heavy network device – into micro-flows (❷ in Figure 4.7) and enqueues each micro-flow's packets onto one selected splitting core (❸ in Figure 4.7). mFlow creates and associates the per-core, per-device splitting queues to the device's NAPI structure `napi_struct` (❶ in Figure 4.7), which can be easily accessed by the network device's softirq handler once executed on the splitting cores (❹ in Figure 4.7).

4.4.2 IRQ-Splitting Function

mFlow implements the IRQ-splitting function in the Mellanox NIC driver – its softirq handler (`mlx5e_napi_poll`). The IRQ-splitting function relies on two inputs from the driver code: a request queue (`mlx5e_rq`), and the way to retrieve requests (`mlx5e_poll_rx_cq`) (❷ in Figure 4.8). With this, mFlow, once enabled, can retrieve any available incoming packet requests in the context of the physical NIC's softirqs and dispatch them onto selected splitting cores (❸ in Figure 4.8). mFlow creates and associates the per-core request buffer to Linux kernel's per-core data structure, `softnet_data`, which can be easily accessed in a softirq context (❶ in Figure 4.8).

mFlow implements the second half (❹ in Figure 4.8) as a regular softirq handler (scheduled by kernel's NAPI scheduler and executed on the splitting cores). The second half can be processed in parallel most of the time except when it needs to update the driver that a packet request has been consumed (*i.e.*, after its `skb` has been created) and can be released (*i.e.*, can be reused for another incoming request). To reduce any possible contention, mFlow updates the driver once in a while (*e.g.*, every 128 requests).

4.4.3 Flow-Reassembling Function

The implementation of batch-based flow reassembling uses two queues – the *backlog* queue for receiving packets from the previous network processing stage and the *receive* queue for delivering packets to user-space applications. Under UDP, `sk_receive_queue` serves as the backlog queue, while `reader_queue` serves as the receive queue. Under TCP, `sk_backlog` serves as the backlog queue, while `sk_receive_queue` serves as the receive queue. mFlow extends the backlog queue into per-core buffer queues (❶ in Figure 4.10), with each serving one splitting core. Thus, all packets from the previous stage are first cached in the buffer queues (❷ in Figure 4.10) before merging. mFlow does not create a new kernel thread for executing the merging functionality. Instead, it adds the merging functionality in the existing kernel thread for packet delivery, *i.e.*, `tcp_recvmsg` for TCP and `udp_recvmsg` for UDP (❸ and ❹ in Figure 4.10). These threads will be woken up when new packets arrive, during which mFlow checks which micro-flow's packets should be merged.

4.5 Evaluation

We have evaluated the effectiveness of mFlow. Results with micro-benchmarks demonstrate that: (1) mFlow significantly improves the throughput of an elephant

single flow – by 81% for TCP and 139% for UDP compared to vanilla overlay networks; (2) mFlow achieves even higher throughput than the native under TCP (29.8 vs. 26.6 Gbps); (3) mFlow reduces average and tail latency for both TCP and UDP. Results with real-world applications demonstrate significant application-level performance benefits brought by mFlow– the performance of a web serving application increases by up to 7.5x, while the latency of a data caching application reduces by up to 48%, compared to vanilla overlay networks.

4.5.1 Experimental Configurations

The experiments were performed on two PowerEdge R740XD servers, each with 2×16-core Intel Xeon Gold 5218 processors (2.30 GHz) and 384 GB memory. The two machines were connected directly by Mellanox ConnectX-5 EN 100-Gigabit Ethernet.

We used Ubuntu 20.04 (with the kernel version 5.7) as the host OSes and the Docker overlay network mode (with Docker version 19.03) as the container overlay network. Docker overlay network uses Linux's builtin VxLAN. We evaluated the following cases: (1) *native*: the physical host network (i.e., no containers); (2) *vanilla overlay*: containers with the default docker overlay network (`VxLAN`); (3) *RPS*: containers with Linux RPS [27] enabled; (4) Falcon: containers with FALCON [64] enabled – the state-of-the-art in-kernel parallelization optimization for container networks; and (5) mFlow.

For mFlow, unless otherwise specified, we set the batch size to 256 and the number of splitting cores to 2, evenly distributed micro-flows to the splitting cores and enabled full path scaling for TCP and device scaling for UDP (in Figure 4.6). For all tests, CPU and memory resources were sufficient. All experiments were run multiple times to mitigate variation.

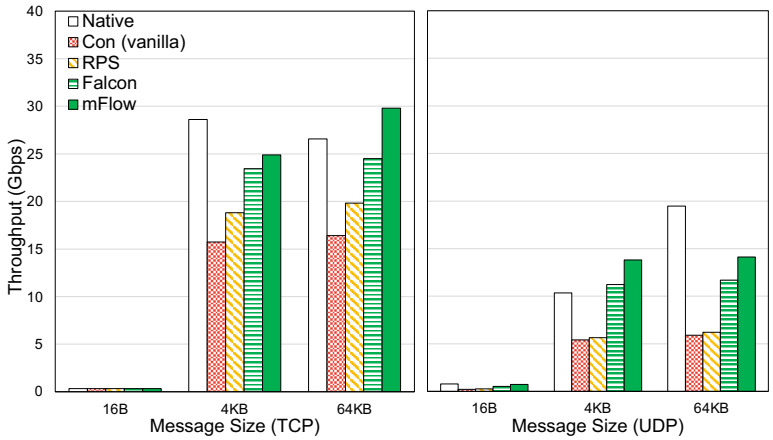

Figure 4.11: Performance comparisons between state-of-the-art approaches.

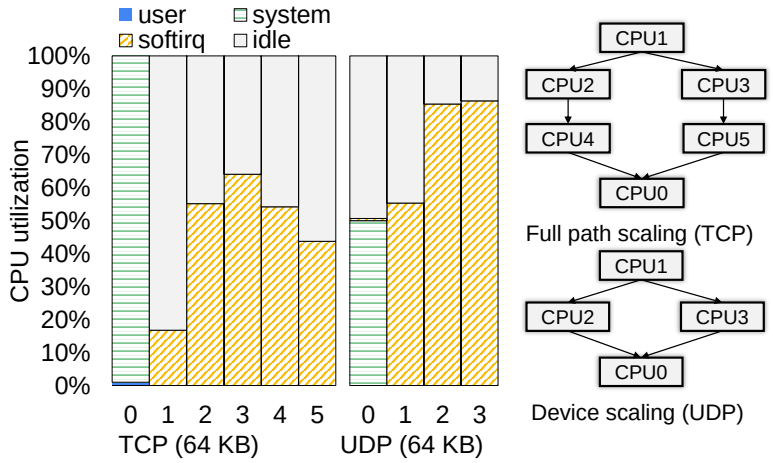

Figure 4.12: CPU utilization breakdown under TCP/UDP with 64KB packet size.

4.5.2 Micro-Benchmarks

Single-flow throughput: To measure the throughput of a single flow, we used `sockperf` [41] to generate traffic with various message sizes. Note that when a message is larger than MTU (1,500 bytes), it will be fragmented into multiple packets during transmission. For TCP, we used a pair of `sockperf` client and server. However, the client under UDP was often bottlenecked (*i.e.*, overloading a CPU core). Hence, similar to FALCON [2], we used *three* `sockperf` clients to send traffic to one

`sockperf` server to stress the network stack on the receiver side to its limit for a UDP flow.

In Figure 4.11, mFlow improved the throughput of a single flow significantly, especially with large message sizes (*e.g.*, 64 KB), by 81% for TCP and 139% for UDP, compared to vanilla overlay. Under TCP, mFlow even achieved higher throughput than the native – 29.8 Gbps vs. 26.6 Gbps. It is because although the native network was much simpler than overlay network, a single core (for `skb_allocation`) was overloaded at the high throughput. In contrast, mFlow leveraged multiple cores to process a single flow in parallel. For UDP (under 64 KB), mFlow achieved lower throughput than the native. The reason is that, under UDP, the clients were throttled after they overloaded client-side CPU cores.

Compared to Falcon, mFlow achieved 22% more throughput under TCP and 21% more under UDP (with 64 KB). It indicates that exploiting *packet-level* parallelism can keep pushing the in-kernel network stack to achieve higher network performance. For UDP under small message/packet size (16B), mFlow achieved even higher performance than Falcon – more than 40%. For TCP with a small packet size (16B), both Falcon and mFlow did not help much (similar to the vanilla overlay). This is because the TCP client became the bottleneck. This also indicates that further optimization focus should be placed on the sender side.

Single-flow splitting and CPU utilization: Figure 4.12 shows how mFlow splits the TCP and UDP flows and the breakdown of average CPU utilization on each core (with 64 KB).

For TCP, we tested mFlow's full path scaling scenario – *i.e.*, splitting occurred in the first stage and merging occurred before packets entered the stateful TCP transport layer. Core *one* was used for dispatching raw packet requests to two separate cores – splitting core *two* and *three*. We noticed that, if all network processings were

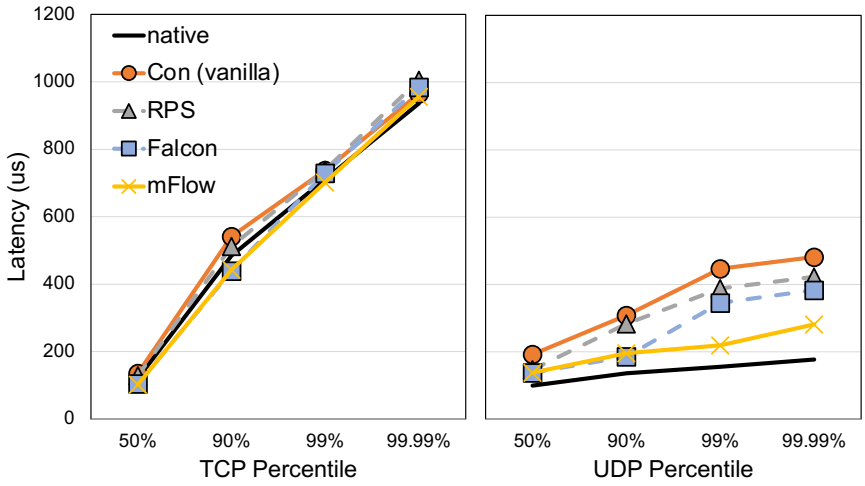

Figure 4.13: Latency comparisons between state-of-the-art approaches and mFlow with 16B message size.

placed on one splitting core, the splitting core was easily overloaded (as mFlow increased TCP throughput significantly). Hence, to scale the performance of a single TCP flow, we further split and pipelined the processings on two cores for each parallel branch – *i.e.*, we used core *two* only for `skb` allocation and dispatched the remaining processings on core *four*. The same configuration was applied to core *three* and *five*. With this, mFlow achieved extremely high TCP throughput for container overlay network as shown in Figure 4.11. Now, we observe that core *zero* – upon which a single kernel thread copies data from the kernel ring buffer to the user-space application – was fully utilized and became the new bottleneck.

For UDP, we tested mFlow's single device scaling scenario – *i.e.*, splitting occurred before the heavyweight `VxLAN` device and merging occurred before packets were copied to applications. As shown in Figure 4.12, we placed all network devices after `VxLAN` on the same core as they consumed way less CPU utilization. Core *one* was used for the first stage and dispatching packets in the form of `skbs` to two separate cores – splitting core *two* and *three*. With this configuration, mFlow achieved

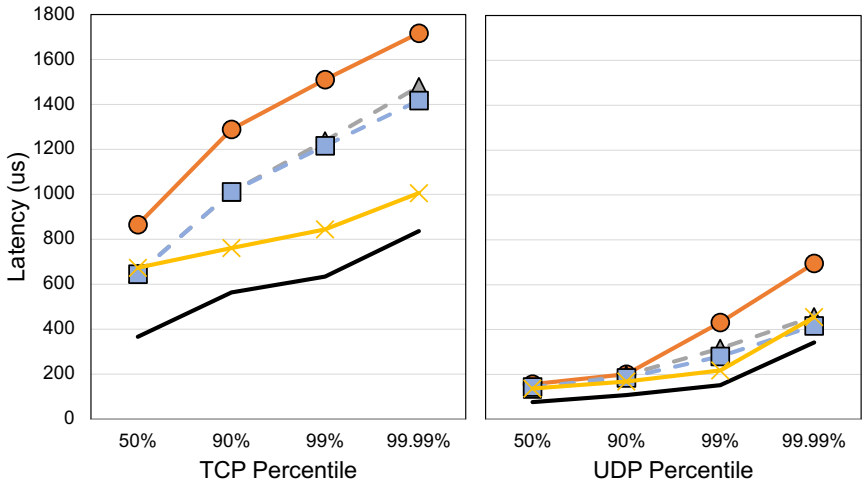

Figure 4.14: Latency comparisons between state-of-the-art approaches and mFlow with 4KB message size.

higher UDP throughput than Falcon for container overlay network (Figure 4.11). We noticed that none of these cores were fully utilized. Instead, the three clients overloaded their sender-side cores and were the bottleneck.

Single-flow latency: Figure 4.13, Figure 4.14 and Figure 4.15 depict the per-packet latency of a single TCP or UDP flow with various message sizes. We measured the latency in the "overloaded" scenario (using sockperf), in which each case was driven to its maximum throughput before packet drops occurred. We observe that, under all cases, mFlow reduced per-packet processing latency compared to vanilla overlay, RPS, and Falcon. For example with 64 KB, compared to vanilla overlay, mFlow reduced the median latency by ∼46% and 99^{th} percentile latency by ∼21% for TCP. It is because mFlow's packet-level parallelism reduces the latency resulting from the pipelined processing (i.e., the processing of the following packet depends on the completion of its previous packet). We observe that there remained a gap in latency between mFlow and the native due to prolonged data path in container overlay networks.

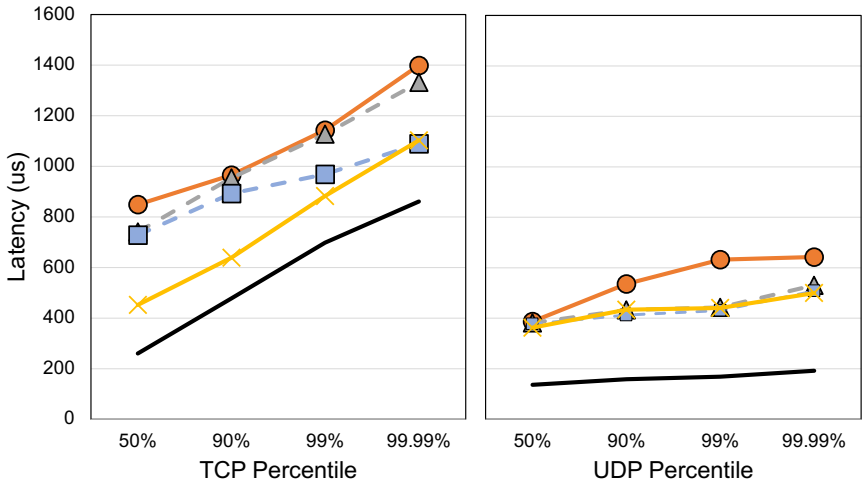

Figure 4.15: Latency comparisons between state-of-the-art approaches and mFlow with 64KB message size.

Multi-flow testing: We further conducted *multi-flow* tests – *i.e.*, multiple flows co-existed within the same host machine. Since for UDP, clients were the main bottlenecks preventing mFlow from saturating available network bandwidth, we showed the multi-flow TCP case in Figure 4.16, Figure 4.17 and Figure 4.18. The message sizes were set to 16 B, 4 KB, and 64 KB, and the number of flows varied from 1 to 20. In all tests, we used 5 dedicated cores for application threads and 10 dedicated cores for all in-kernel packet processing to have a more controlled experimental environment for the ease of result analysis.

In Figure 4.16, Figure 4.17 and Figure 4.18, with the small message/packet size (*i.e.*, 16 B), all test cases scaled linearly, as the client side became the bottleneck. With the larger message/packet sizes (*i.e.*, 4 KB and 64 KB), mFlow consistently outperformed vanilla overlay – *e.g.*, by 24% with 5 concurrent flows (under 4 KB). This benefit shrank as more flows were added – *e.g.*, by 11% with 10 flows and by 5% under 20 flows. It is because as the flow number increased, there was little CPU resource to scale up mFlow. This can be further verified with the comparison

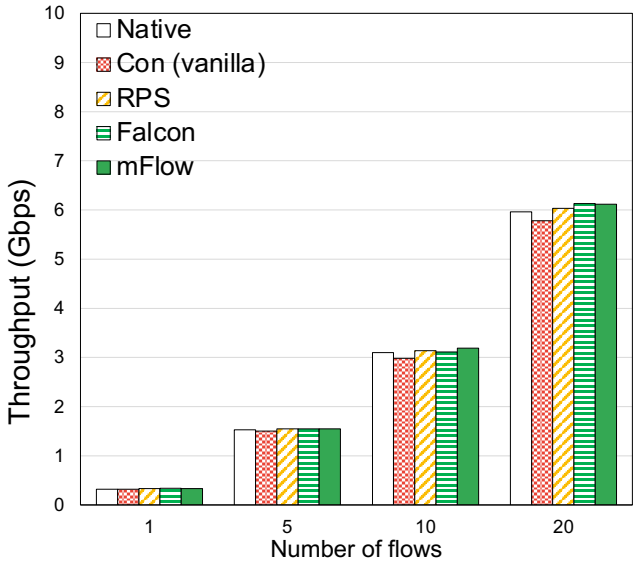

Figure 4.16: Accumulated network throughput with multiple TCP flows with 16B packet size.

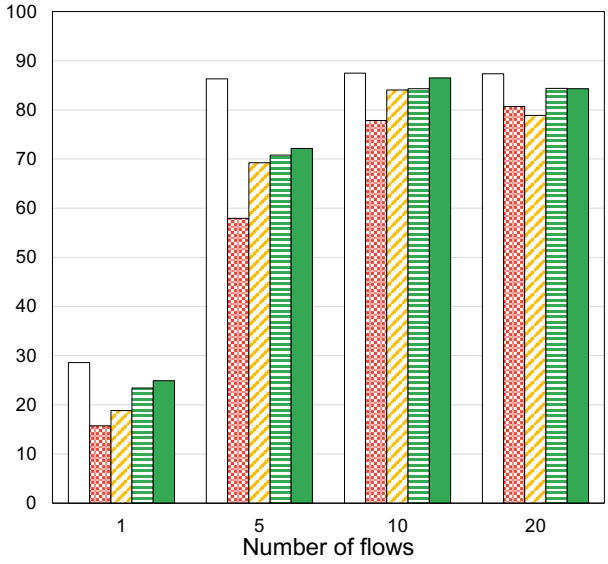

Figure 4.17: Accumulated network throughput with multiple TCP flows with 4KB packet size.

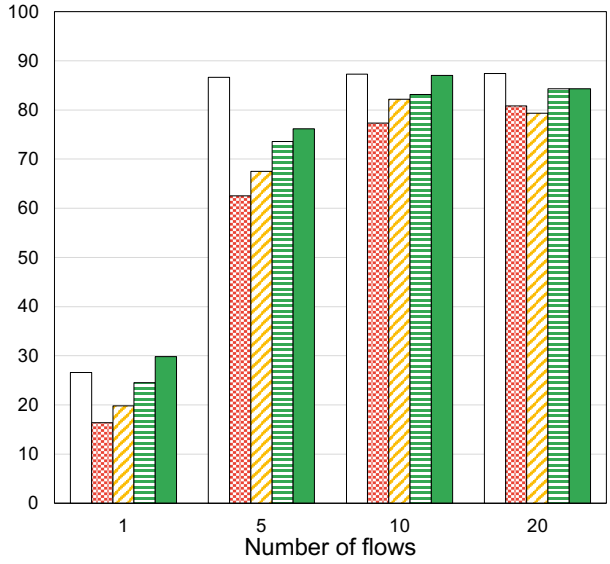

Figure 4.18: Accumulated network throughput with multiple TCP flows with 64KB packet size.

between FALCON and MFLOW – MFLOW outperformed FALCON by 5% with 10 concurrent flows (with 64 KB) while they achieved the same performance with 20 flows, where CPU was the bottleneck.

mFlow overhead: Figure 4.19 shows the average CPU load distribution among all used cores for the multiple TCP flow case (with 10 flows under 64 KB). More fine-grained flow steering in MFLOW does incur additional overhead – compared to FALCON, MFLOW consumed 15% more CPU utilization (among 10 cores for packet processing) in exchange for 5% performance gains. However, this is the worst-case scenario. We observed less than 5% additional overhead with 5 flows and the same CPU utilization with 20 flows (the system was overloaded). On the other hand, the advantage of MFLOW lies in that, in Figure 4.19, MFLOW can leverage CPU cores in a more balanced manner with even load distribution. In contrast, CPU utilization variation under FALCON was larger than MFLOW – *i.e.*, the standard deviation of CPU utilization among 10 cores was 20.5 (FALCON) vs. 11.6 (MFLOW).

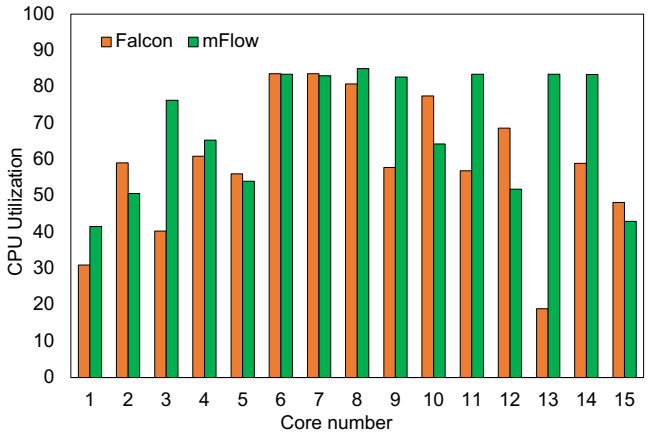

Figure 4.19: mFlow uses CPU cores in a more balanced manner.

4.5.3 Applications

In this section, we use two representative real-world applications, web serving and data caching, to evaluate mFlow.

Web serving: We measured the performance of Cloudsuite's Web Serving benchmark [48] under vanilla overlay, Falcon, and mFlow. Cloudsuite's Web Serving – the benchmark to evaluate page load throughput and access latency – contains four components: an *nginx* web server, a *mysql* database, a *memcached* server, and clients. The web server runs the Elgg [49] social network and connects to the cache and database servers. The clients send different types of request workloads, including login, chat, update, etc., to the web server. In our experiments, all of the services were performed inside containers that were connected via the Docker overlay network upon the 100 Gbps NIC.

Figure 4.20 depicts the "success operation" rate when we ran the benchmark with 200 users. We observe that mFlow improved the successful individual operation rate by 2.3x – 7.5x compared to the vanilla overlay network. For the same metric, mFlow outperformed Falcon by 1.5x – 3.6x. Figure 4.21 and Figure 4.22 present the average response time and delay time for different operations. The response

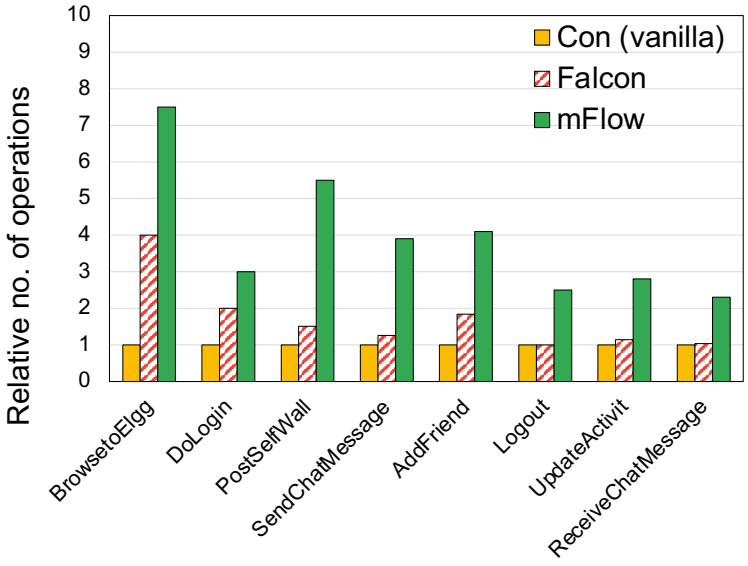

Figure 4.20: mFlow improves success operation of a web serving application.

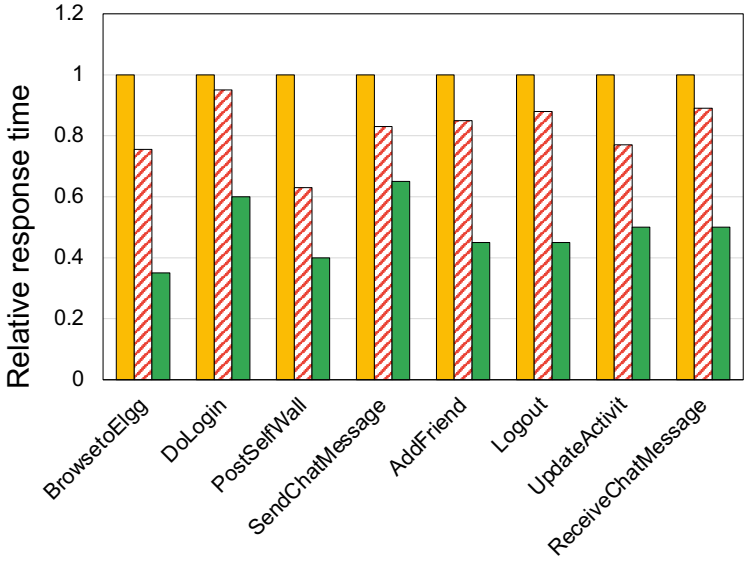

Figure 4.21: mFlow reduces average response time of a web serving application.

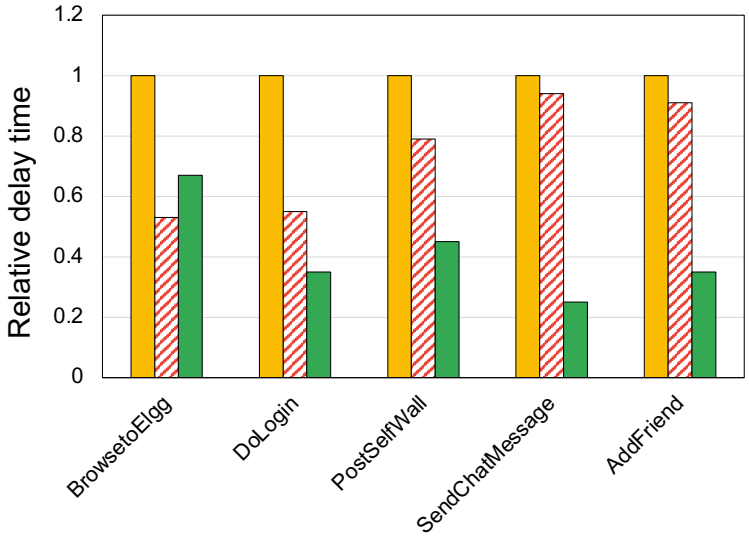

Figure 4.22: mFlow reduces average delay time of a web serving application.

time denotes the time to complete one request while the delay time represents the difference between the target and actual processing time. Compared to the vanilla overlay network, mFlow reduced the average response time by 35% – 65% while the average delay time by up to 75%. Compared to Falcon, mFlow reduced the average response time by 22% – 54% and the average delay time by 36% – 73%.

Data caching: We further measured the average and tail latency using Cloudsuite's data caching benchmark. It uses the *Memcached* data caching server, simulating the behavior of a Twitter caching server with a Twitter dataset. In our experiments, the Memcached server was configured with 4GB memory, 4 threads, and 550 bytes object size. The Memcached server and clients were running under the same Docker overlay network. As illustrated in Figure 4.23, compared to the vanilla overlay network, mFlow reduced the tail latency (99^{th} percentile latency) by 26% when we used one client. When the number of clients increased to ten, mFlow's benefit became more significant – reducing the average and tail latency by 48% and 47% (99^{th} percentile). It is because, as the number of clients (and the request rate) increased,

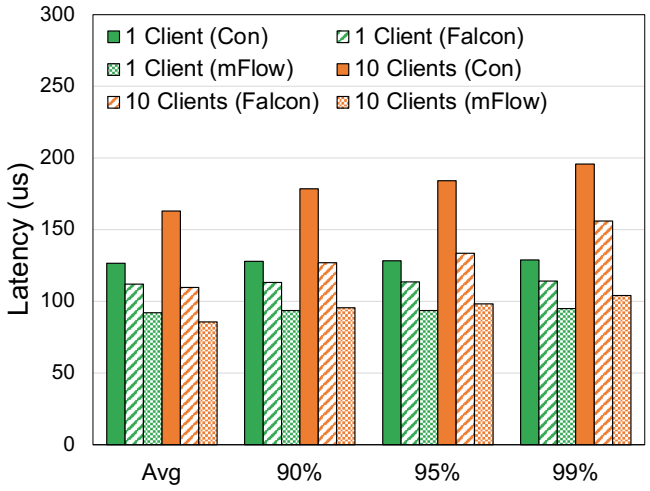

Figure 4.23: MFLOW reduces the average and tail latency of a data caching application (Memcached).

the in-kernel network stack was more stressed. MFLOW improved its efficiency by using multiple cores for parallel packet processing. In addition, compared to FALCON, MFLOW reduced the average latency by 22% and tail latency (99^{th} percentile) by 33%, demonstrating a higher degree of packet processing parallelism.

4.6 Conclusion

We have presented MFLOW, a novel in-kernel packet steering approach to accelerate container overlay networks by exploiting packet-level parallelism. MFLOW splits the packets of a single flow into multiple micro-flows and processes them in parallel by taking advantage of a multi-core system while efficiently preserving in-order packet delivery. Our evaluation with both micro-benchmarks and applications demonstrates the effectiveness of MFLOW. Meanwhile, the results have revealed new bottlenecks that prevent a single flow from further scaling: One lies in clients/senders and the other is the receiver-side single data-copying thread. We seek to address these bottlenecks in our future work.

CHAPTER 5

RELATED WORK

5.1 Network Stack Optimization

Researchers have identified that poor network performance often stems from inefficiencies and complexities within the kernel network stack. Consequently, numerous studies have proposed various optimizations along the data path, addressing issues such as interrupt processing [22, 65, 66, 24, 67], system call batching [68], protocol processing [22, 69], memory copying [22, 23, 24, 25], scheduling [65, 70, 24, 71, 72, 73], and application-kernel interaction [74, 75]. Despite these efforts, some work indicates that both latency and throughput remain significantly below hardware capabilities [76, 24]. Unlike these traditional network improvements, our work focuses on optimizing specific issues within container networks, making these studies orthogonal to ours.

Beyond existing OS enhancements, other research has proposed lightweight and customized network stacks [76, 77, 78, 79, 80, 81, 14, 82] to boost network performance. For instance, Slim [14] is a connection-oriented approach that creates overlay networks by manipulating connection metadata. While containers use private IPs for connections, packets use host IPs for transmission. Slim bypasses virtual bridges and network devices in containers, achieving near-native performance. However, Slim does not support connection-less protocols like UDP and complicates host network management, as each overlay connection requires a unique file descriptor and port. Some designs require changes to the application-kernel interface, rendering them incompatible with legacy applications. Alternatively, research has shifted to bypassing the OS kernel entirely, implementing the network stack in user space [21, 79, 47]. User-space

approaches reduce context switches and allow direct hardware access, minimizing kernel indirection and overhead. Intel's Data Plane Development Kit (DPDK) [21] is one example of such user-space libraries. Our work aims to enhance the efficiency of commodity OS kernels to support all network traffic in overlay networks while retaining existing network management tools.

5.2 Kernel Scalability on Multicore

As the number of CPU cores increases, improving resource utilization, system efficiency, scalability, and concurrency has become a hot research topic. Boyd-Wickizer *et al.* [83] analyzed the scalability of applications running on Linux atop a 48-core machine, revealing that nearly all applications encounter scalability bottlenecks within the Linux kernel. Many researchers advocate rethinking operating systems [84, 85], proposing new kernels for high scalability, such as Barrelfish [86] and Corey [87]. The availability of multiple processors in computing nodes and multiple cores in a processor has also motivated proposals to utilize multicore hardware, including protocol onloading or offloading on dedicated processors [88, 89, 90, 91], network stack parallelization [92, 93, 94, 95], packet processing alignment [96, 97], and optimized scheduling [98, 99, 97]. However, none of these techniques specifically address inefficiencies within container networks. Our work focuses on mitigating the serialization of softirq execution due to overlay networks in the Linux kernel.

5.3 Container Network Acceleration

Container networks, being a new and complex technology, suffer from various inefficiencies. To diagnose bottlenecks and optimize container networks, many techniques have been developed recently. These works fall into two categories: First,

many researchers propose to reduce unnecessary work to improve the performance. Systems can offload CPU-intensive tasks, such as checksum computation, onto hardware [100, 101, 102, 103] or bypass inefficient kernel parts [104, 103] to improve container network processing. Advanced offloading techniques, like Mellanox ASAP2 [40], enable virtual switches and packet transformation to be handled entirely by NIC hardware, achieving near-native overlay performance. However, this approach has drawbacks: it is only available on high-end hardware, restricts overlay network configuration flexibility and scalability, and requires SR-IOV to pass virtual functions (VFs) directly to containers, limiting the number of containers per host (e.g., 512 VFs in the Mellanox ConnectX®-5 100 Gbps Ethernet adapter) [105]. Another category of works, including virtual routing [10], memory sharing [39], resource management [106], redistribution and reassignment [107], and manipulating connection-level metadata [14]. Unlike these works, our research addresses the inefficiencies in interrupt processing within container networks. We propose solutions leveraging multicore hardware with minimal modifications to the kernel stack and data plane.

CHAPTER 6

FUTURE WORK

My future research will continue to develop innovative solutions to further enhance the efficiency and performance of cloud networking infrastructures, by leveraging techniques that include kernel optimization, user space offloading, and the integration of reconfigurable, disaggregated hardware.

6.1 **SmartNIC-Assisted Zero-Copying**

Through our efforts working in system kernel, the efficiency and scalability of container overlay networks have experienced significant improvements. However, a substantial performance gap remains when comparing a single flow throughput to the physical bandwidth of hardware NICs (e.g., 100 Gbps). Our research reveals the final-step processing in the kernel as an inevitable bottleneck, specifically due to the overhead of copying packet payloads from kernel to user space. Overcoming this requires addressing two key challenges not yet solved by existing methods: First, only the packet payloads should be accessible to user applications, which implies the necessity to separate the packet headers and payloads during processing. Second, the existing kernel-user communication interface, which relies on data copying, requires an overhaul with a newly implemented zero-copying interface.

In response, we plan to develop a full-stack zero-copying framework which orchestrates hardware, kernel, and user applications to enhance networking performance by eliminating the copying overhead. Within this framework, packet headers and payloads are separated (based on identifying the protocol header or computing the

header length) on SmartNICs (e.g., Nvidia BlueField [108]) as a pre-processing stage as packets approach the host server. Upon arrival at the host server, it is the kernel's responsibility to allocate distinct memory regions for storing the headers and payloads separately. Headers undergo further processing via the traditional kernel protocol stack, while payloads are held in a designated memory area, awaiting access by user application via mapping-based zero-copying. Once the headers have been processed in the kernel network stack, user applications can retrieve the packet payloads via a new zero-copying interface. To ensure compatibility with legacy applications, we will deploy a new dynamic runtime library that allows the applications to be launched with the new interface without requiring any modifications.

6.2 **Elastic Networking Offloading**

Existing hardware offloading solutions of network stack are still limited. Consider two typical types of SmartNICs: SoC-based SmartNICs offer limited resources [109]. Developers can offload only part of the network stack [110]. FPGA-based SmartNICs provide minimal flexibility. Network protocols are hard-coded into the hardware, making protocol adaptation to new applications needs a time-consuming process of hardware reprogramming [111]. Our objective is to develop elastic solutions that integrate SmartNICs with host servers, thereby addressing the complexities of network traffic processing and optimizing cloud networking offloading efficiency.

This research comprises two primary components. Firstly, we aim to develop a transparent and elastic offloading scheme by following a modular design principle. This approach separates the stateless packet processing path into multiple fine-grained modules (or pipelines), encompassing both low-level network functions such as packet partitioning and transport, and high-level computation tasks including filtering, sorting, encapsulation/decapsulation. Given that different devices have varied processing

capabilities and different types of traffic impose distinct pressures to networking modules, a crucial feature is allowing for the flexible integration of these modules across NICs, host server kernels, or even user-space libraries. For example, packet processing involving OVS-related packet filters (e.g., TC flower [112]), packet classification [113], and overlay encapsulation/decapsulation can be extensively offloaded to SmartNICs via ASAP2 technology [114] to leverage ASIC-based hardware accelerators for enhanced performance. Secondly, we plan to deploy a central scheduler that monitors the computation resources and memory constraints on the SmartNIC. This scheduler dynamically reallocates computation workloads between the host CPU and the SmartNIC in real-time, ensuring optimal utilization of resources in both of them to prevent the slower SmartNIC cores from becoming a bottleneck.

6.3 **Reshaping Networking via CXL**

Emerging applications and an increasingly huge amount of data are driving a greater specialization in hardware. As a result, the PCIe link [115], which connects the CPU to these devices, is becoming a bottleneck. Compute Express Link (CXL) is a cache-coherent interconnect for processors, memory expansion, and specialized accelerators [116]. Although its developement is still in early stages, CXL's potential to transform the underlying cloud architecture has already attracted our attention. Our research will focus primarily on the benefits of utilizing CXL to build a shared memory system among specialized hardware (e.g., SmartNICs) and host machines to accelerate networking.

Firstly, CXL enables coherent cache and consistent memory access between CPUs and peripheral devices. This capability can expand the limited memory embedded in specialized hardware, significantly enhancing functional scalability. For instance, TC flower serves as a packet filter and classifier, running on the SmartNIC

with its control plane located in the host server. In a cloud multi-tenancy architecture, connections can expand to millions. Deploying traffic rules, which involve copying the rules from the host server memory to the NIC memory, can consume substantial CPU resources. Additionally, the limited size of NIC memory can restrict the ability of TC to serve large-scale services. With CXL, a SmartNIC can directly access the host's large amount of memory, supporting millions of rules while eliminating the overhead of copying rules from host memory to NIC memory. Secondly, utilizing SmartNICs as intermediary nodes allows for the offloading of all CXL-related computations. In the cloud, many applications that span across cores or require cooperative workloads face the challenge of data movement between cores. To meet these needs, future SmartNICs will be crucial in offloading data movements between containers, VMs, accelerators, and CXL attached memory. This offloading improves core efficiency, enforces security and access policies, and provides advanced features such as remote atomics used for statistics and other purposes [117]. Lastly, CXL interconnects are expected to be more efficient than traditional networks. The reason is that traditional networking requires conversions between network protocols and the PCIe protocol. For instance, a NIC uses PCIe to communicate with its host, but network interactions typically use the TCP/IP software stack or RDMA. In contrast, all traffic in CXL already travels in CXL messages with unified semantics, which should result in lower latency and higher bandwidth. With these insights and observations, a lot of great opportunities are already exposed to us in investigating the new cloud infrastructures with reconfigurable, disaggregated harware.

CHAPTER 7

CONCLUSION

In this dissertation, I have systematically explored and addressed the efficiency and scalability challenges of container overlay networks in cloud environments. To tackle these challenges, I made several significant and innovative contributions:

Firstly, our comprehensive performance study on multi-core systems with high-speed network devices identified three critical parallelization bottlenecks within the kernel network stack: (1) the lack of effective parallelization mechanism in system kernel, preventing a single container flow from achieving high network throughput; (2) inefficient handling of various packet processing tasks, preventing multiple container flows from saturating a 40 Gbps network link; and (3) these issues become more severe with small packets due to the kernel's inability to manage a large number of interrupts, disrupting overall system efficiency. Secondly, We proposed Falcon, a fast and balanced container networking approach which parallelizes softirq processing within a single network flow. Falcon employs three key designs: softirq pipelining, splitting, and dynamic balancing, enabling low-cost flow parallelization on multicore machines. Our experimental results demonstrated that Falcon significantly enhances the performance of container overlay networks, as evidenced by improvements in both micro-benchmarks and real-world applications. Thirdly, We presented mFlow, a novel in-kernel packet steering approach which leverages fine-grained, packet-level parallelism by splitting packets of a single flow into multiple micro-flows and processing them in parallel across multiple cores while preserving in-order packet delivery.

Our evaluations with micro-benchmarks and applications showcased the effectiveness of mFlow.

Through these contributions, I have demonstrated significant improvements in the efficiency and scalability of cloud networking systems, specifically the container overlay networks. My research provides a solid foundation for further enhancements in cloud-based container networking, paving the way for more efficient and scalable cloud infrastructures.

REFERENCES

[1] J. Lei, K. Suo, H. Lu, and J. Rao, "Tackling parallelization challenges of kernel network stack for container overlay networks," in *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, 2019, pp. 9–9.

[2] J. Lei, M. Munikar, K. Suo, H. Lu, and J. Rao, "Parallelizing packet processing in container overlay networks," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 261–276.

[3] J. Lei, M. Munikar, H. Lu, and R. Jia, "Accelerating packet processing in container overlay networks via packet-level parallelism," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 79–89.

[4] *Google Cloud Container*, https://cloud.google.com/containers.

[5] *Apache Mesos*, https://mesos.apache.org.

[6] *Kubernetes*, https://kubernetes.io.

[7] *Docker Swarm*, https://docs.docker.com/engine/swarm/.

[8] *Flannel*, https://github.com/flannel-io/flannel.

[9] *Weave*, https://github.com/weaveworks/weave.

[10] *Calico*, https://github.com/projectcalico/calico.

[11] *Use Overlay Networks*, https://docs.docker.com/network/drivers/overlay/.

[12] *VxLAN*, https://datatracker.ietf.org/doc/html/rfc7348.

[13] *Open vSwitch (OVS)*, https://www.openvswitch.org.

[14] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim: Os kernel support for a low-overhead container overlay

network," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, 2019, pp. 331–344.

[15] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 189–197.

[16] L. Angrisani, L. Peluso, A. Tedesco, and G. Ventre, "Measurement of processing and queuing delays introduced by a software router in a single-hop network," in *2005 IEEE Instrumentationand Measurement Technology Conference Proceedings*, vol. 3. IEEE, 2005, pp. 1797–1802.

[17] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 15–28.

[18] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet," in *Proceedings of the 5th annual Linux Showcase & Conference-Volume 5*, 2001, pp. 18–18.

[19] *GRO: Surviving 10Gbp/s with Cycles to Spare*, https://events.static.linuxfound.org/images/stories/slides/jls09/jls09_xu.pdf.

[20] *Large Receive Offload (LRO)*, https://lwn.net/Articles/243949/.

[21] *DPDK*, https://www.dpdk.org.

[22] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt, "Performance analysis of system overheads in tcp/ip workloads," in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. IEEE, 2005, pp. 218–228.

[23] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems

for the cloud," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 461–472, 2013.

[24] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 4, pp. 1–30, 2015.

[25] L. Rizzo, "netmap: a novel framework for fast packet i/o," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[26] *Receive Side Scaling (RSS)*, https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[27] *Receive Packet Steering (RPS)*, https://lwn.net/Articles/362339/.

[28] *Linux Overlay Network*, https://www.kernel.org/doc/Documentation/networking/vxlan.txt.

[29] *Docker*, https://www.docker.com.

[30] *Consul*, https://www.consul.io.

[31] *iPerf3*, https://iperf.fr.

[32] *Packet Length Distributions*, https://www.caida.org/catalog/papers/2000_aix0005/aix0005/.

[33] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2015, pp. 171–172.

[34] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th international middleware conference*, 2016, pp. 1–13.

[35] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *2014 IEEE International Conference on Cloud Engineering*. IEEE, 2014, pp. 610–614.

[36] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[37] *8 surprising facts about real docker adoption*, https://www.datadoghq.com/docker-adoption/.

[38] K. Suo, Y. Zhao, W. Chen, and J. Rao, "vnettracer: Efficient and programmable packet tracing in virtualized networks," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 165–175.

[39] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar, "Freeflow: High performance container networking," in *Proceedings of the 15th ACM workshop on hot topics in networks*, 2016, pp. 43–49.

[40] *OVS Offload Using ASAP2 Direct*, https://docs.nvidia.com/networking/display/mlnxofedv473290/ovs+offload+using+asap2+direct.

[41] *Sockperf*, https://github.com/Mellanox/sockperf.

[42] *Flame Graph*, https://github.com/brendangregg/FlameGraph.

[43] *Encrypting Network Traffic*, https://encryptionhowto.sourceforge.net/Encryption-HOWTO-5.html.

[44] *TCPdump*, https://www.tcpdump.org.

[45] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 163–174, 2014.

[46] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE communications magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[47] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014, pp. 459–473.

[48] *CloudSuite*, https://www.cloudsuite.ch.

[49] *Elgg*, https://elgg.org.

[50] *Memcached*, https://memcached.org.

[51] N. Zhou, Y. Georgiou, M. Pospieszny, L. Zhong, H. Zhou, C. Niethammer, B. Pejak, O. Marko, and D. Hoppe, "Container orchestration on hpc systems through kubernetes," *Journal of Cloud Computing*, vol. 10, pp. 1–14, 2021.

[52] C. Cérin, N. Grenèche, and T. Menouer, "Towards pervasive containerization of hpc job schedulers," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 281–288.

[53] G. Li, J. Woo, and S. B. Lim, "Hpc cloud architecture to reduce hpc workflow complexity in containerized environments," *Applied Sciences*, vol. 11, no. 3, p. 923, 2021.

[54] Y. Zhou, B. Subramaniam, K. Keahey, and J. Lange, "Comparison of virtualization and containerization techniques for high performance computing," in *Proceedings of the 2015 ACM/IEEE conference on Supercomputing*, 2015.

[55] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, G. Morgan, and R. Ranjan, "A study on the evaluation of hpc microservices in containerized environ-

ment," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 7, pp. 1–1, 2021.

[56] S. Herbein, A. Dusia, A. Landwehr, S. McDaniel, J. Monsalve, Y. Yang, S. R. Seelam, and M. Taufer, "Resource management for running hpc applications in container clouds," in *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*. Springer, 2016, pp. 261–278.

[57] A. Ruhela, M. Vaughn, S. L. Harrell, G. J. Zynda, J. Fonner, R. T. Evans, and T. Minyard, "Containerization on petascale hpc clusters." State of Practice talk in International Conference for High Performance . . . , 2020.

[58] A. Torrez, T. Randles, and R. Priedhorsky, "Hpc container runtimes have minimal or no performance impact," in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE, 2019, pp. 37–42.

[59] M. Höb and D. Kranzlmüller, "Enabling easey deployment of containerized applications for future hpc systems," in *Computational Science–ICCS 2020: 20th International Conference, Amsterdam, The Netherlands, June 3–5, 2020, Proceedings, Part I 20*. Springer, 2020, pp. 206–219.

[60] *Separation Anxiety: A Tutorial for Isolating Your System with Linux Namespaces*, https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces.

[61] *Linux Control Groups*, https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html.

[62] *Seccomp security profiles for Docker*, https://docs.docker.com/engine/security/seccomp/.

[63] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin, "Is network the bottleneck of distributed training?" in *Proceedings of the Workshop on Network Meets AI & ML*, 2020, pp. 8–13.

[64] *Falcon*, https://github.com/munikarmanish/falcon.

[65] L. Cheng and C.-L. Wang, "vbalance: Using interrupt load balance to improve i/o performance for smp virtual machines," in *Proceedings of the third ACM symposium on cloud computing*, 2012, pp. 1–14.

[66] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "Eli: Bare-metal performance for i/o virtualization," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 411–422, 2012.

[67] *Performance Tuning for Mellanox Adapters*, https://enterprise-support.nvidia.com/s/article/performance-tuning-for-mellanox-adapters.

[68] L. Soares and M. Stumm, "Flexsc: flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010, pp. 33–46.

[69] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck, "An in-depth study of lte: Effect of network protocol and application behavior on performance," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 363–374, 2013.

[70] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 431–442, 2014.

[71] K. Suo, Y. Zhao, J. Rao, L. Cheng, X. Zhou, and F. C. Lau, "Preserving i/o prioritization in virtualized oses," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 269–281.

[72] Y. Zhao, K. Suo, L. Cheng, and J. Rao, "Scheduler activations for interference-resilient smp virtual machine scheduling," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017, pp. 222–234.

[73] Y. Zhao, K. Suo, X. Wu, J. Rao, S. Wu, and H. Jin, "Preemptive multi-queue fair queuing," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 147–158.

[74] P. Emmerich, D. Raumer, A. Beifuß, L. Erlacher, F. Wohlfart, T. M. Runge, S. Gallenmüller, and G. Carle, "Optimizing latency and cpu load in packet processing systems," in *2015 International symposium on performance evaluation of computer and telecommunication systems (SPECTS)*. IEEE, 2015, pp. 1–8.

[75] H. Huang, J. Rao, S. Wu, H. Jin, K. Suo, and X. Wu, "Adaptive resource views for containers," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 243–254.

[76] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: a protected dataplane operating system for high throughput and low latency," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, 2014, pp. 49–65.

[77] Y. Huang, J. Geng, D. Lin, B. Wang, J. Li, R. Ling, and D. Li, "Los: A high performance and compatible user-level network operating system," in *Proceedings of the First Asia-Pacific Workshop on Networking*, 2017, pp. 50–56.

[78] M. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mos: a reusable networking stack for flow monitoring middleboxes," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, 2017, pp. 113–129.

[79] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level tcp stack for multicore systems," in *Pro-*

ceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, 2014, pp. 489–502.

[80] Z. Niu, H. Xu, D. Han, P. Cheng, Y. Xiong, G. Chen, and K. Winstein, "Network stack as a service in the cloud," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017, pp. 65–71.

[81] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, 2012, pp. 61–72.

[82] M. Munikar, J. Lei, H. Lu, and J. Rao, "Prism: Streamlined packet processing for containers with flow prioritization," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022, pp. 336–346.

[83] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010, pp. 1–16.

[84] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[85] D. Patterson, "The parallel revolution has started: are you part of the solution or part of the problem? an overview of research at the berkeley parallel computing laboratory," in *International Conference on High Performance Computing for Computational Science*. Springer, 2010, pp. 26–27.

[86] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os ar-

chitecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 29–44.

[87] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, *et al.*, "Corey: an operating system for many cores," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 43–57.

[88] P. Gilfeather and A. B. Maccabe, "Modeling protocol offload for message-oriented communication," in *2005 IEEE International Conference on Cluster Computing*. IEEE, 2005, pp. 1–10.

[89] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong, "Tcp onloading for data center servers," *Computer*, vol. 37, no. 11, pp. 48–58, 2004.

[90] P. Shivam and J. S. Chase, "On the elusive benefits of protocol offload," in *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, 2003, pp. 179–184.

[91] R. Westrelin, N. Fugier, E. Nordmark, K. Kunze, and E. Lemoine, "Studying network protocol offload with emulation: approach and preliminary results," in *Proceedings. 12th Annual IEEE Symposium on High Performance Interconnects.* IEEE, 2004, pp. 84–90.

[92] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley, "Performance issues in parallelized network protocols," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, 1994, pp. 10–es.

[93] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 121–136.

[94] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: taking the v out of nfv," in *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, 2016, pp. 203–216.

[95] P. Willmann, S. Rixner, and A. L. Cox, "An evaluation of network stack parallelization strategies in modern operating systems," in *Proceedings of the annual conference on USENIX'06 Annual Technical Conference*, 2006, pp. 8–8.

[96] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, "Improving network connection locality on multicore systems," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 337–350.

[97] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 325–341.

[98] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: preemptive scheduling for μsecond-scale tail latency," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, 2019, pp. 345–359.

[99] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: achieving high cpu efficiency for latency-sensitive datacenter workloads," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, 2019, pp. 361–377.

[100] *Improving Overlay Solutions with Hardware-Based VXLAN Termination*, https://www.pica8.com/wp-content/uploads/pica8-whitepaper-VXLAN-overlay.pdf.

[101] *Mellanox VXLAN Acceleration*, https://www.slideshare.net/slideshow/vxlan-vm-world-02-theatre/38833540.

[102] *Optimizing the Virtual Network with VXLAN Overlay Offloading*, https://www.intel.com/content/www/us/en/developer/overview.html#gs.b681eb.

[103] J. Weerasinghe and F. Abel, "On the cost of tunnel endpoint processing in overlay virtual networks," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE, 2014, pp. 756–761.

[104] *Scalable High-Performance User Space Networking for Containers*, https://www.dpdk.org/wp-content/uploads/sites/35/2016/08/Day02-Session02-Steve-Liang-DPDKUSASummit2016.pdf.

[105] *World-Class Performance Ethernet SmartNICs Product Line*, https://network.nvidia.com/files/doc-2020/pb-bluefield-smart-nic.pdf.

[106] Y. Hu, M. Song, and T. Li, "Towards" full containerization" in containerized network function virtualization," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 467–481.

[107] Y. Zhang, Y. Li, K. Xu, D. Wang, M. Li, X. Cao, and Q. Liang, "A communication-aware container re-distribution approach for high performance vnfs," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 1555–1564.

[108] *NVIDIA BlueField Networking Platform*, https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[109] X. Wei, R. Cheng, Y. Yang, R. Chen, and H. Chen, "Characterizing off-path {SmartNIC} for accelerating distributed systems," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 987–1004.

[110] T. Kim, D. M. Ng, J. Gong, Y. Kwon, M. Yu, and K. Park, "Rearchitecting the {TCP} stack for {I/O-Offloaded} content delivery," in *20th USENIX Sympo-*

sium on Networked Systems Design and Implementation (NSDI 23), 2023, pp. 275–292.

[111] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, *et al.*, "Azure accelerated networking:{SmartNICs} in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 51–66.

[112] *TC Flower*, https://netdevconf.info/2.2/papers/horman-tcflower-talk.pdf.

[113] P. Wang, F. Wen, P. V. Gratz, and A. Sprintson, "Simd-matcher: A simd-based arbitrary matching framework," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 3, pp. 1–20, 2022.

[114] *ASAP2: Accelerated Switching and Packet Processing*, https://network.nvidia.com/files/doc-2020/sb-asap2.pdf.

[115] *PCIe*, https://pcisig.com.

[116] *Compute Express Link (CXL)*, https://computeexpresslink.org.

[117] *AMD SmartNIC with the support of CXL*, https://www.amd.com/content/dam/amd/en/documents/products/accelerators/alveo/adaptive-smartnic-white-paper.pdf.

BIOGRAPHICAL STATEMENT

Jiaxin Lei was born in Xi'an, Shaanxi, China, in 1994. He obtained a B.E. in Telecommunication Engineering with Management from Beijing University of Posts and Telecommunications (BUPT) in 2017. He received an M.S. in Computer Science from the State University of New York (SUNY) at Binghamton in 2018. He completed his Ph.D. in Computer Science at The University of Texas at Arlington (UTA) in 2024. Throughout his doctoral studies, he focused on developing efficient and scalable cloud networking systems, with a particular emphasis on container overlay networks. His research interests include cloud computing, networking systems, reconfigurable hardware, and datacenter infrastructure.