Parallelizing Packet Processing in Container Overlay Networks

Jiaxin Lei[*]
Binghamton University
jlei23@binghamton.edu

Manish Munikar[*]
The University of Texas at Arlington
manish.munikar@uta.edu

Kun Suo[*]
Kennesaw State University
ksuo@kennesaw.edu

Hui Lu
Binghamton University
huilu@binghamton.edu

Jia Rao
The University of Texas at Arlington
jia.rao@uta.edu

Abstract

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 paper 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).

ACM Reference Format:
Jiaxin Lei, Manish Munikar, Kun Suo, Hui Lu, and Jia Rao. 2021. Parallelizing Packet Processing in Container Overlay Networks. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456241

1 Introduction

Due to its high performance [38, 66], low overhead [36, 68], and widespread community support [53], container technology has increasingly been adopted in both private data centers and public clouds. A recent report from Datadog [1] 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 [11]. 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 [2], Kubernetes [13], and Docker Swarm [6].

Container networks provide connectivity to distributed applications and are critical to building large-scale, container-based services. Overlay networks, e.g., Flannel [10], Weave [26], Calico [4] and Docker overlay [25], are widely adopted in container orchestrators [3, 6, 13]. 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 [50, 68, 69, 74, 78].

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 [78] 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) [20], focus on

[*]Equal contribution.

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 such data path, the state-of-the-art seeks to either eliminate packet transformation from the network stack [78] or offload the entire virtual switches and packet transformation to the NIC hardware[19]. 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 [78] does not apply to connection-less protocols, while advanced hardware offloading is only available in high-end hardware [19].

This paper 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 measurements and profiling, 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 provides a dynamic balancing mechanism to effectively multiplex softirqs of multiple flows on a multi-core system for efficient interrupt processing.

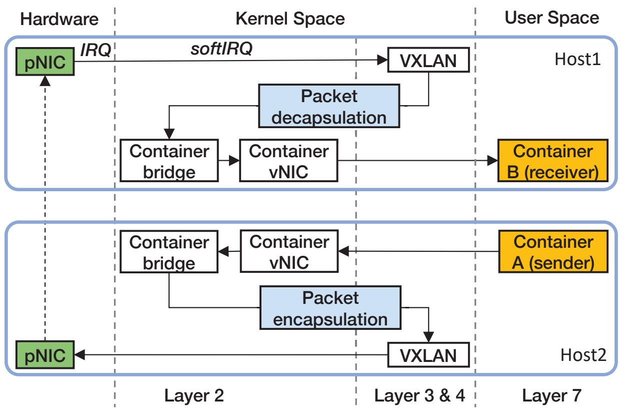


Figure 1. Illustration of container overlay networks.

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 paper 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).

Road map: Section 2 discusses the background and gives motivating examples of the comparison between a physical network and a container overlay network. Section 3 analyzes reasons causing the degradation in container network performance. Section 4 presents design details of FALCON while Section 5 provides its implementation. Section 6 shows the experimental results. Section 7 reviews related works and Section 8 concludes the paper.

2 Background and Motivation

In this section, we first describe the process of packet processing in OS kernel. Then, we examine the performance overhead 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.

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 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 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 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 networking devices, such as bridges, virtual Ethernet ports (vNIC), routers, etc., to support the connectivity of containers across multiple hosts. Indeed, 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 of 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) [21] 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) [20] 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.

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. Details on the software and hardware configurations can be found in Section 6.

Single-flow throughput. Figure 2 depicts the performance loss due to the overlay network in various settings. Figure 2 (a) shows the comparison between overlay and host networks in a throughput stress test. We used *sockperf* [23] 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 [78], 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 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

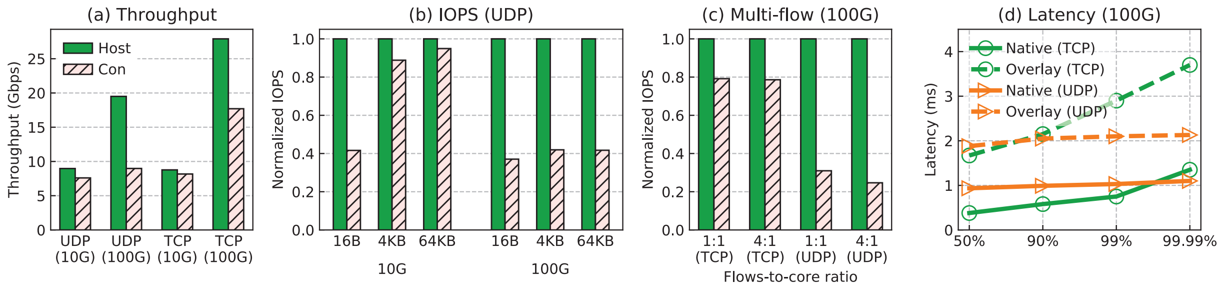


Figure 2. The performance comparison of container overlay network and the native physical network.

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 single flows 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 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 flows, 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 single 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 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 underlying network and packet rate. In what follows, we analyze the root causes of overlay-induced performance degradation.

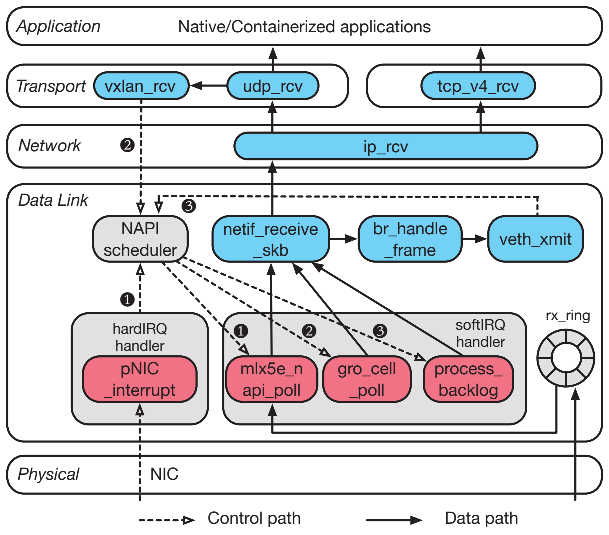


Figure 3. Packet reception in a container overlay network.

3 Root Cause Analysis

3.1 Prolonged Data Path

We draw the call graph of packet reception in the Linux kernel using `perf` and `flamegraph` [9] and analyze the control and data paths in the host and overlay networks. As Figure 3 illustrates, packet reception in an overlay network involves 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

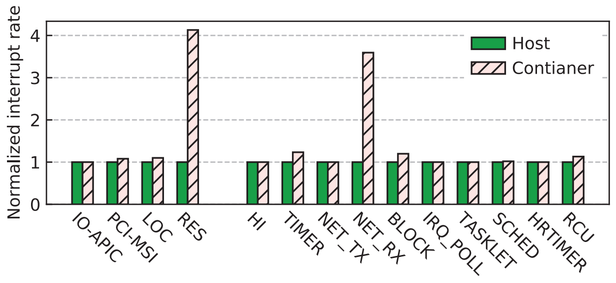


Figure 4. The comparison of hardware and software interrupt rates in the native and container overlay networks.

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 [16]) 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.

3.2 Excessive, Expensive, and Serialized Softirqs

Call graph analysis suggests that overlay networks invoke more softirqs than the native network does. Figure 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.

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) that is raised by the CPU scheduler attempting to spread load across multiple cores. Figure 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.

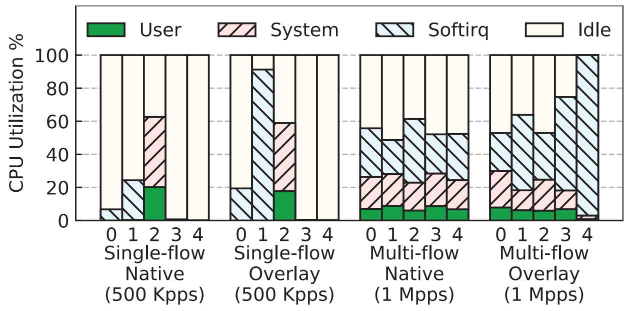


Figure 5. Serialization of softirqs and load imbalance.

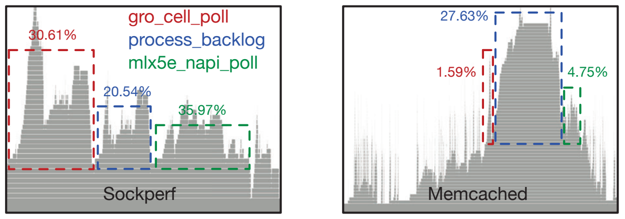


Figure 6. Flamegraphs of Sockperf and Memcached.

Not only are there more softirqs in overlay networks, some of them become more expensive than that in the native network. Figure 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 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 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 [78], 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.

4 Falcon 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 7), as detailed as follows.

4.1 Software Interrupt Pipelining

Inspired by RPS [20], 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.

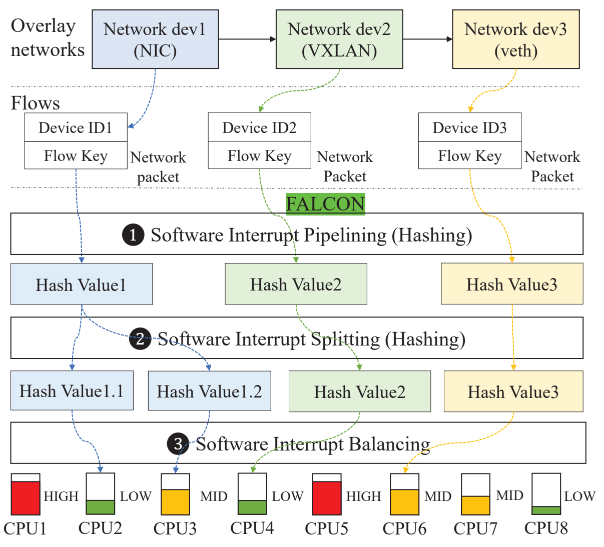


Figure 7. Architecture of Falcon. Falcon consists of three main techniques:❶ software interrupt pipelining, ❷ software interrupt splitting, and ❸ software interrupt balancing.

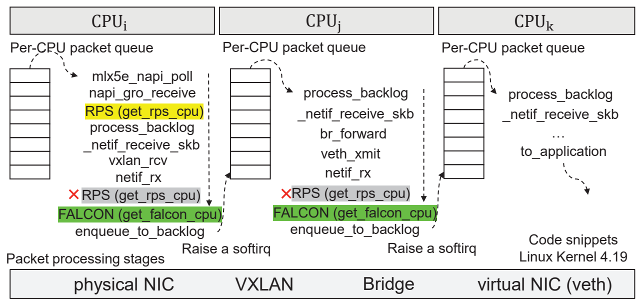


Figure 8. Falcon pipelines software interrupts of a single flow by leveraging stage transition functions.

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), 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 8, though the RPS function is invoked multiple times along the network path, only the first RPS (on CPU$_i$) takes effect (i.e., selecting a new CPU core based on the hash value), while the following RPS (e.g., on CPU$_i$ and on CPU$_j$) generate the same hash value for the packets of the same flow.

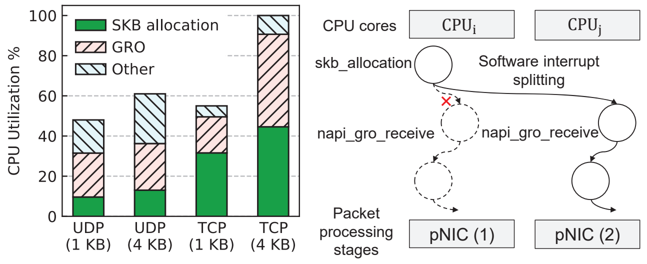
A natural way to distinguish different processing stages of a single flow is to involve additional device information for the hash calculation: We notice that, when a packet is sent or received by a new network device, the device pointer (dev) in the packet's data structure (sk_buff) will be updated 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 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 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) for separation of network processing, it coexists with existing scaling techniques like RPS/RSS.

4.2 Software Interrupt Splitting

Though it makes intuitive sense to separate network processing stages at *per-device* granularity (in Section 4.1), 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.



(a) CPU% of first stage. **(b)** Software interrupt splitting.

Figure 9. (a) A single device takes up a single core under TCP with large packet size (4 KB). (b) Falcon splits the processing of a "heavy" network device into multiple smaller tasks with each running on a separate core.

As plotted in Figure 9a, 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 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 and 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 [8], profiling [24], in-kernel software switches [17], and many network functions [39, 42, 52].

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 9b, 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 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

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

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

into two, with each for a half processing of the associated network device (e.g., `pNIC1` and `pNIC2` in Figure 9b).

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].

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 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` (see Section 6.1 for parameter sensitivity). 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).

[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.

Algorithm 1 Dynamic Softirq Balancing

```
 1: Variables: socket buffer skb; current average load of the sys-
    tem $L_{avg}$; network flow hash skb.hash and device ID ifindex;
    Falcon CPU set FALCON_CPUS.
 2:
 3: // Stage transition function
 4: function NETIF_RX(skb)
 5:     // Enable Falcon only if there is room for parallelization
 6:     if $L_{avg}$ < FALCON_LOAD_THRESHOLD then
 7:         cpu := get_falcon_cpu(skb)
 8:         // Enqueue skb to cpu's packet queue and raise softirq
 9:         enqueue_to_backlog(skb, cpu)
10:     else
11:         // Original execution path (RPS or current CPU)
12:         ...
13:     end if
14: end function
15:
16: // Determine where to place the next softirq
17: function GET_FALCON_CPU(skb)
18:     // First choice based on device hash
19:     hash := hash_32(skb.hash + ifindex)
20:     cpu := FALCON_CPUS[hash % NR_FALCON_CPUS]
21:     if cpu.load < FALCON_LOAD_THRESHOLD then
22:         return cpu
23:     end if
24:     // Second choice if the first one is overloaded
25:     hash := hash_32(hash)
26:     return FALCON_CPUS[hash % NR_FALCON_CPUS]
27: end function
```

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 dynamic balancing algorithm is inspired by comprehensive experimentation with container networks and the network stack. The central design is the use of hash-based, two random choices in CPU selection. As CPU load cannot be accurately measured at a per-packet level, we observed significant fluctuations in CPU load due to frequent softirq migrations that aggressively seek to find the least loaded CPU. On the other hand, purely random balancing based on device hash may lead to persistent hotspots. The two-choice algorithm avoids long-lasting hotspots by steering away from a busy CPU in the first attempt but commits to the second choice in order to minimize load fluctuations.

5 Implementation

We have implemented Falcon upon Linux network stack in two generations of Linux kernel, 4.19 and 5.4, with the focus on the presented three components as stated in Section 4.

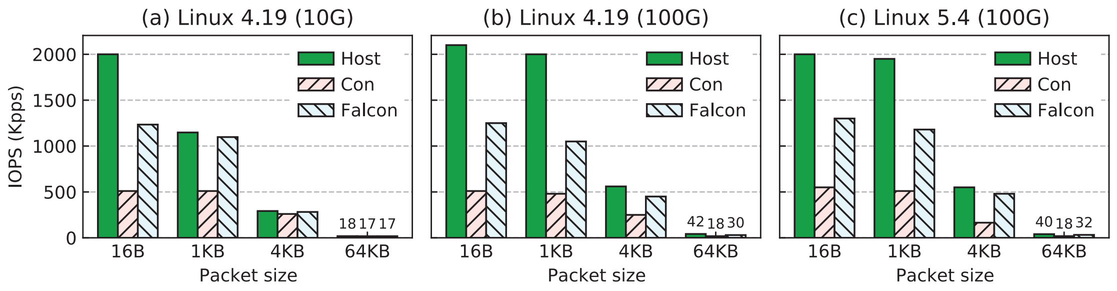


Figure 10. Packet rates in the host network, vanilla overlay, and Falcon overlay under a UDP stress test.

Falcon is available at: https://github.com/munikarmanish/falcon. Underpinning Falcon's implementation, there are two specific techniques:

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.

Hashing mechanism. As stated in Section 4.3, 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.

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 (Section 6.1), 2) significantly improves latency for both UDP and TCP (Section 6.1), and 3) achieves even higher than native throughput in multi-flow TCP tests (Section 6.1). 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 (Section 6.2). Nevertheless, overhead analysis (Section 6.3) 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.

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*).

6.1 Micro-benchmarks

Single-flow stress test. As shown in Figure 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 10 shows that FALCON achieved significant throughput improvements over Docker overlay, especially with large packet sizes. It delivered near-native 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 11 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.

Single-flow latency. Figure 12 depicts per-packet latency in single-flow UDP and TCP tests. We are interested in latency in both 1) underloaded tests, wherein client sending

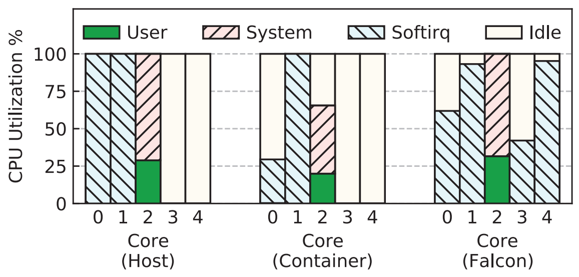


Figure 11. CPU utilization of a single UDP flow.

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 12 (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 12 (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 12 (b) and (d) shows the effect of FALCON on TCP latency. Our experiments found that in the overloaded TCP test (Figure 12 (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 12 (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.

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 CPU and memory resources were sufficient (i.e., not the bottleneck). 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 13 (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

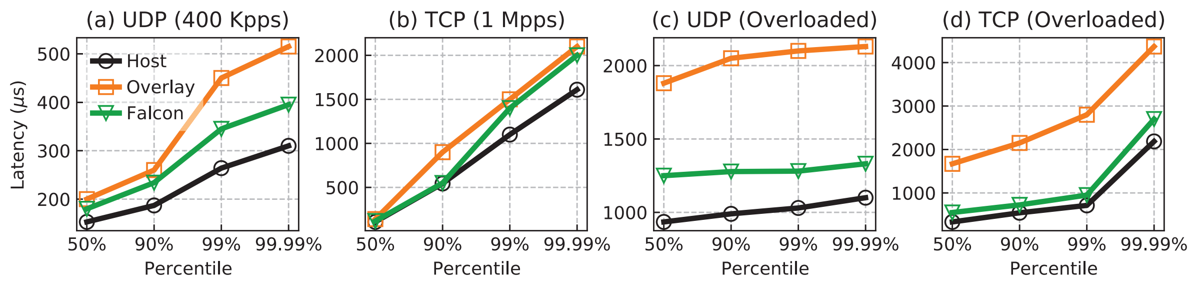


Figure 12. Effect of Falcon on per-packet latency. Packet size is 16 B in (a, c, d) and 4 KB in (b).

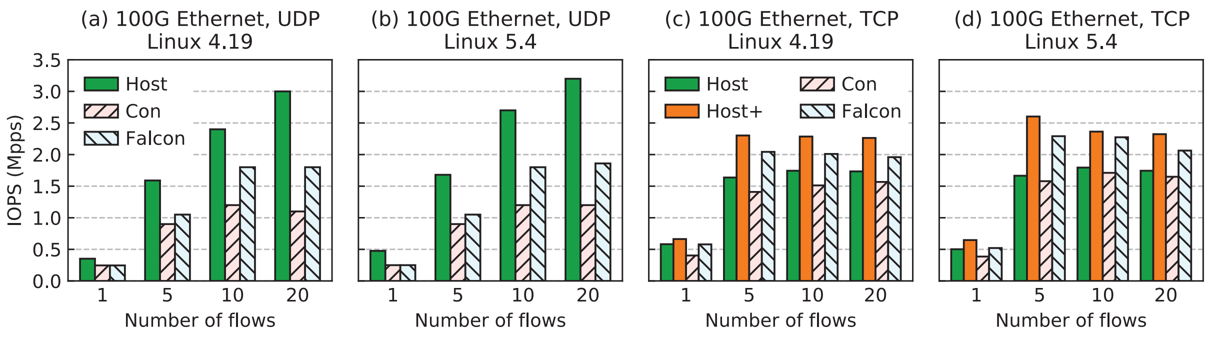


Figure 13. Packet rates in the host network, vanilla overlay, and Falcon under multi-flow UDP and TCP tests.

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 13 (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%.

Multi-container throughput. This sections compares Falcon with existing packet steering techniques (i.e., RSS/RPS) in *multi-container* tests — multiple containers were running with each hosting one flow. Unlike the above multi-flow tests with sufficient CPU resources, the CPU resources were saturated when the number of containers was high enough under the multi-container tests. As illustrated in Figure 14, we gradually saturated 6 cores [4] with the increasing number of containers from 6 to 40. We observe that: 1) when the system has idle CPU resources (e.g., under 6 or 10 containers), Falcon is able to improve overall throughput (by up to 27% under UDP and 17% under TCP); 2) when the system is fully saturated (e.g., 100% utilization with 20 and more containers), Falcon achieves the same throughput as RSS/RPS. It is because, Falcon improves performance when there are idle CPU resources for parallelization. However, when the system is fully loaded, Falcon's dynamic balancing algorithm (Section 4.3) will effectively disable Falcon to prevent any performance loss due to overhead — as will be discussed below (Section 6.3), Falcon does incur (a little) overhead due to additional softirq migration.

[4]It was impractical for us to saturate a 40-core system due to limited client machines; hence we used 6 dedicated cores.

Parameter sensitivity. As seen in the above multi-container tests, Falcon can be dynamically switched on/off when the system load is low/high. It is determined by a configurable threshold, `FALCON_LOAD_THRESHOLD`. In this section, we test the effect of the threshold and its impact on container network performance. We evaluated the performance of Falcon with varying thresholds on a variety of workloads — same as the multi-container tests. From Figure 15, we observe that always enabling Falcon (i.e., the always-on case) hurts the network performance when the system is highly loaded. Similarly, having a threshold lower than 70% would waste a lot of CPU cycles. We can see that a threshold between 80–90% results in the best performance in practice.

Adaptability test. To measure the effectiveness of Falcon's *dynamic* balancing algorithm (Section 4.3), we tested a scenario where some flows suddenly increase their load from low to high. In a statically hashed algorithm (e.g., RSS/RPS), the softirq-to-core mapping, once determined, never changes. So when a lightweight flow abruptly becomes heavyweight and overloads a core, it cannot adapt to the changing workload and the overall performance suffers. On the other hand, with Falcon's *two-choice* dynamic re-balancing approach, some softirqs from the overloaded core are remapped to less-loaded cores, thereby removing the bottleneck and increasing the overall throughput. Figure 16 clearly demonstrates this effect: We first balanced 6 flows onto 6 cores and then suddenly increased the packet rate in one of them to overload

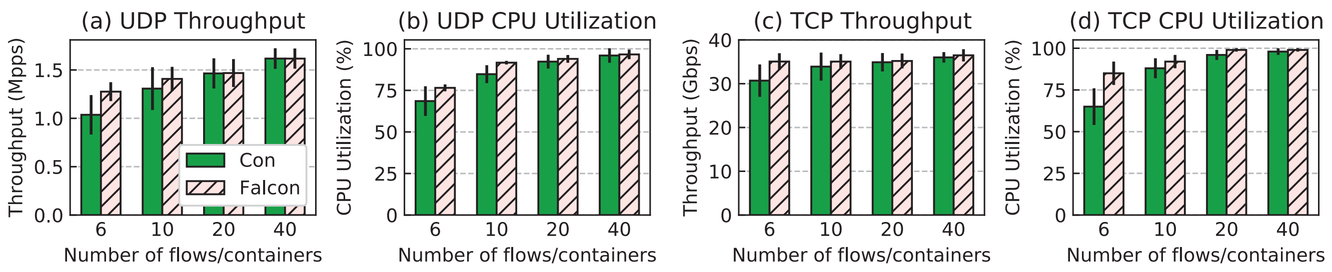


Figure 14. FALCON does not degrade the performance when the system is overloaded with many small flows.

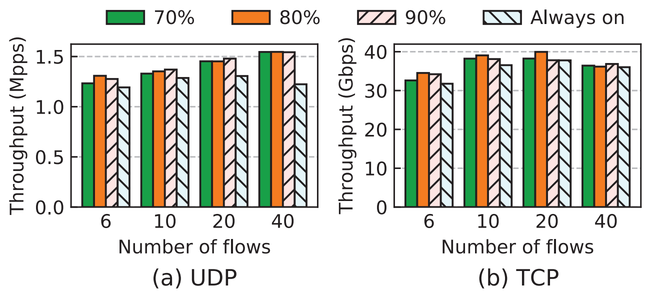


Figure 15. Effect of the average load threshold and its impact on container network performance.

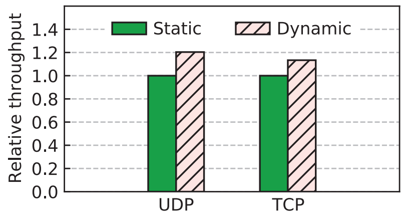


Figure 16. FALCON adapts to changing workload and rebalances softirqs, dynamically.

a core. We compare the throughput of this dynamically imbalanced workload under static hashing vs. dynamic rehashing (FALCON's two-choice algorithm). As expected, FALCON achieves 18% improvement in the UDP throughput and about 15% improvement in the TCP throughput, compared to the static hashing (with only one fixed choice).

6.2 Application Results

Web serving. We measured the performance of the Cloudsuite's Web Serving benchmark [5] 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 [7] 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 17(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 17(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.

Data caching. We further measured the average and tail latency using Cloudsuite's data caching benchmark, *memcached* [15]. 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 18, 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, as stated in Section 6.4.

6.3 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

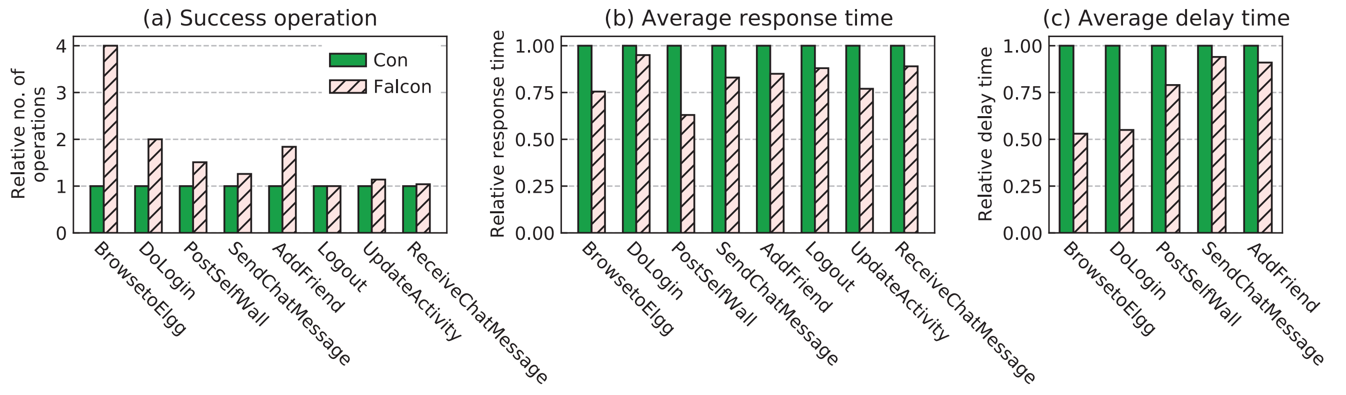


Figure 17. FALCON improves the performance of a web serving application (from Cloudsuite) in terms of higher operation rate and lower response time, compared to vanilla overlay network.

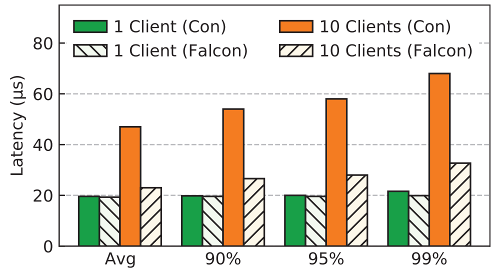


Figure 18. FALCON reduces the average and tail latency under data caching using Memcached.

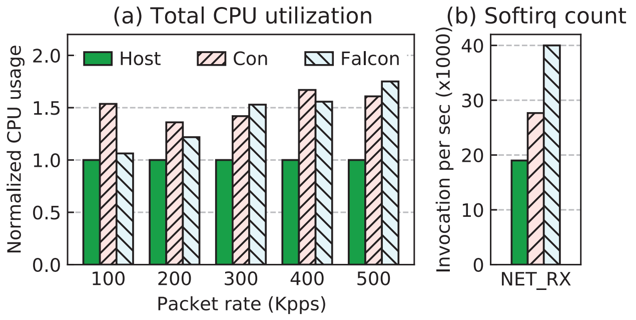


Figure 19. Overhead of FALCON.

CPU usage given the same traffic load, we quantify it by measuring the total CPU usage with fixed packet rates. Figure 19 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 19(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 19(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. It is because FALCON involves more network devices and needs more CPU resources for packet processing in the overlay network.

6.4 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 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.

[5]Note that, the overlay network triggers fewer softirqs in Figure 19(b) (1.45x of the native) than that in Figure 4 (3.6x of the native), as we measured it in a less loaded case (400 Kpps).

7 Related Work

Network stack optimization. Many researchers have revealed that the poor performance of network suffered from the inefficiency and complexity inside the kernel network stack. Therefore, lots of studies and solutions have been proposed to optimize problems along the data path, including the interrupt processing [31, 34, 41, 61], protocol processing [31, 45], memory copying [31, 51, 61, 64], scheduling [34, 35, 61, 70, 76, 77], interaction between the application and kernel [37, 44], etc. Different from the above work improving the traditional network, our work focuses on optimizing the issues existed specifically inside the container networks and thus the above studies are orthogonal to ours. In addition to renovating the existing OSes, some other papers proposed lightweight and customized network stacks [30, 46–48, 55, 65, 78] to improve the network performance. For example, Slim [78] is a connection-oriented approach that creates overlay networks by manipulating connection metadata. Containers can still use private IPs to establish connections but packets use host IPs for transmission. In Slim, network virtualization is realized via connection redirection at the connection level rather than packet transformation at the packet level. As such, Slim can bypass the virtual bridge and the virtual network device in containers, achieving near-native performance. However, Slim does not apply to connection-less protocols, such as UDP, and complicates and limits the scalability of host network management since each overlay network connection needs a unique file descriptor and port in the host network. In this work, we strive to salvage a commodity OS kernel to efficiently support all network traffic in overlay networks.

Kernel scalability on multicore. As the number of CPU core increases, how to improve the resource utilization and the system efficiency, scalability and concurrency is becoming a hot research topic. Boyd-Wickizer *et al.* [33] analyzed the scalability of applications running on Linux on top of a 48-core machine and reported almost all applications triggered scalability bottlenecks inside the Linux kernel. Many researchers advocated rethinking the operating systems [28, 59] and proposed new kernel for high scalability, such as Barrelfish [29] and Corey [32]. The availability of multiple processors in computing nodes and multiple cores in a processor also motivated proposals to utilize the multicore hardware, including protocol onloading or offloading on dedicated processors [40, 63, 67, 72], network stack parallelization [54, 57, 58, 73], packet processing alignment [60, 62], optimized scheduling [49, 56, 62], to improve the network performance. However, none of the above techniques are designed on optimizing the inefficiency inside container networks. Instead, FALCON addresses the serialization of softirq execution due to overlay networks in Linux kernel.

Container network acceleration. As a new and complex technique, many reasons could contribute to inefficiency of container networks. In order to diagnose bottlenecks and optimize container networks, many researches and techniques have been developed in recent years. These works can be divided in two categories. First, many researchers propose to reduce unnecessary work to improve the performance. Systems can offload CPU-intensive work, such as checksum computing, onto hardwares [12, 14, 18, 71] or bypass inefficient parts inside kernel [22, 71] to improve the container network processing. As a concrete example, advanced offloading techniques, e.g., Mellanox ASAP2 [19], allow for the offloading of virtual switches and packet transformation entirely to the NIC hardware. This technique helps deliver near-native overlay performance as packets coming out of the NIC are stripped off host network headers and can be processed as ordinary packets in physical networks. However, it has several drawbacks: 1) advanced offloading is only available in high-end hardware; 2) it has restrictions on the configuration of overlay networks, limiting flexibility and scalability. For example, SR-IOV has to be enabled to directly pass virtual functions (VFs) to containers as a network device. This not only increases the coupling of containers with the hardware but also limits the number of containers in a host, e.g., 512 VFs in the Mellanox ConnectX®-5 100 Gbps Ethernet adapter [27]. Another category of works, including virtual routing [4], memory sharing [74], resource management [43], redistribution and reassignment [75], manipulating connection-level metadata [78], focus on optimizing the data path along container networks. Different from above works, our work focuses on the inefficiency of interrupt processing inside container networks and proposes solutions to address them by leveraging the multicore hardware with little modification to the kernel stack and data plane.

8 Conclusions

This paper has demonstrated that the performance loss in overlay networks due to serialization in the handling of excessive, expensive softirqs can be significant; the interrupt processing in container overlay networks should be scaled to leverage multicore. We introduce FALCON, a fast and balanced container networking approach, which mitigates the overhead of overwhelming and unbalanced software interrupts in the container overlay networks. FALCON decomposes the packet processing of distinct network devices for a single flow along the data path, and multiplexes and balances software interrupts of multiple flows among multiple cores effectively. Our experimental results show that FALCON can significantly improve the performance of container overlay networks with both micro and real-world applications.

9 Acknowledgments

We thank our shepherd, Michio Honda, and the anonymous reviewers for their insightful comments. This work was supported by NSF under Award 1909877 and 1909486.

References

[1] *8 surprising facts about real docker adoption.* https://goo.gl/F94Yhn.
[2] *Apache Mesos.* http://mesos.apache.org/.
[3] *Apache Mesos.* https://mesos.apache.org/.
[4] *Calico.* https://github.com/projectcalico/calico-containers.
[5] *cloudsuite.* https://cloudsuite.ch.
[6] *Docker Swarm.* https://docs.docker.com/engine/swarm/.
[7] *Elgg.* https://elgg.org.
[8] *Encrypting Network Traffic.* http://encryptionhowto.sourceforge.net/Encryption-HOWTO-5.html.
[9] *Flame Graph.* https://github.com/brendangregg/FlameGraph.
[10] *Flannel.* https://github.com/coreos/flannel/.
[11] *Google Cloud Container.* https://cloud.google.com/containers/.
[12] *Improving Overlay Solutions with Hardware-Based VXLAN Termination.* https://goo.gl/5sV8s6.
[13] *Kubernetes.* https://kubernetes.io/.
[14] *Mellanox VXLAN Acceleration.* https://goo.gl/QJU4BW.
[15] *Memcached.* https://memcached.org/.
[16] *Open vSwitch.* https://www.openvswitch.org/.
[17] *Open vSwitch.* http://openvswitch.org/.
[18] *Optimizing the Virtual Network with VXLAN Overlay Offloading.* https://goo.gl/LEquzj.
[19] *OVS Offload Using ASAP2 Direct.* https://docs.mellanox.com/display/MLNXOFEDv471001/OVS+Offload+Using+ASAP2+Direct.
[20] *Receive Packet Steering.* https://lwn.net/Articles/362339/.
[21] *Receive Side Scaling.* https://goo.gl/BXvmAJ.
[22] *Scalable High-Performance User Space Networking for Containers.* https://goo.gl/1SJjro.
[23] *Sockperf.* https://github.com/Mellanox/sockperf.
[24] *TCPDump.* https://www.tcpdump.org/.
[25] Use overlay networks. https://docs.docker.com/network/overlay/.
[26] *Weave.* https://github.com/weaveworks/weave.
[27] World-Class Performance Ethernet SmartNICs Product Line. https://www.mellanox.com/files/doc-2020/ethernet-adapter-brochure.pdf.
[28] 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*, 2009.
[29] 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 architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, 2009.
[30] 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 USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2014.
[31] 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 *Proceedings of 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2005.
[32] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, et al. Corey: An operating system for many cores. In *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.
[33] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich, et al. An analysis of linux scalability to many cores. In *Proceedings of 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
[34] 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*, page 2. ACM, 2012.
[35] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.
[36] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *Proceedings of IEEE IC2E*, 2014.
[37] 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 *Proceedings of International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, 2015.
[38] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *Proceedings of IEEE ISPASS*, 2015.
[39] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM Computer Communication Review*, 2014.
[40] P. Gilfeather and A. B. Maccabe. Modeling protocol offload for message-oriented communication. In *Proceedings of the IEEE International Cluster Computing*, 2005.
[41] 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. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.
[42] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 2015.
[43] Y. Hu, M. Song, and T. Li. Towards full containerization in containerized network function virtualization. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
[44] 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 (HPDC)*, 2019.
[45] 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. In *Proceedings of ACM SIGCOMM*, 2013.
[46] 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 (APNet)*, 2017.
[47] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mos: A reusable networking stack for flow monitoring middleboxes. In *Proceedings of USENIX NSDI*, 2017.
[48] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[49] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI})*, 2019.
[50] J. Lei, K. Suo, H. Lu, and J. Rao. Tackling parallelization challenges of kernel network stack for container overlay networks. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, 2019. USENIX Association.
[51] 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. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
[52] 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 USENIX NSDI*, 2014.
[53] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. In *Linux Journal*, 2014.

[54] 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 (OSDI)*, 1994.

[55] 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 (HotNets)*, 2017.

[56] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI})*, 2019.

[57] 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 (SOSP)*, 2015.

[58] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the V out of NFV. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[59] D. Patterson. The parallel revolution has started: Are you part of the solution or part of the problem? In *International Conference on High Performance Computing for Computational Science (SC)*, 2010.

[60] 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 (Eurosys)*, 2012.

[61] 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. 2014.

[62] 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 (SOSP)*, 2017.

[63] 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. *IEEE Computer*, 2004.

[64] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *Proceedings of 21st USENIX Security Symposium (USENIX Security)*, 2012.

[65] 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 (CoNEXT)*, 2012.

[66] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of ACM Middleware*, 2016.

[67] 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.

[68] K. Suo, Y. Zhao, W. Chen, and J. Rao. An analysis and empirical study of container networks. In *Proceedings of IEEE INFOCOM*, 2018.

[69] K. Suo, Y. Zhao, W. Chen, and J. Rao. vNetTracer: Efficient and programmable packet tracing in virtualized networks. In *Proceedings of IEEE ICDCS*, 2018.

[70] 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 Symposium on Cloud Computing (SoCC)*, 2017.

[71] J. Weerasinghe and F. Abel. On the cost of tunnel endpoint processing in overlay virtual networks. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*, 2014.

[72] R. Westrelin, N. Fugier, E. Nordmark, K. Kunze, and E. Lemoine. Studying network protocol offload with emulation: approach and preliminary results. In *Proceedings of 12th IEEE Symposium on High Performance Interconnects (HOTI)*, 2004.

[73] P. Willmann, S. Rixner, and A. L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2006.

[74] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar. Freeflow: High performance container networking. In *Proceedings of ACM HotNet*, 2016.

[75] 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 *Proceedings of IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[76] Y. Zhao, K. Suo, L. Cheng, and J. Rao. Scheduler activations for interference-resilient smp virtual machine scheduling. In *Proceedings of the ACM/IFIP/USENIX Middleware Conference (Middleware)*, 2017.

[77] Y. Zhao, K. Suo, X. Wu, J. Rao, S. Wu, and H. Jin. Preemptive multiqueue fair queuing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019.

[78] D. Zhuo, K. Zhang, Y. Zhu, H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. Slim: OS kernel support for a low-overhead container overlay network. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.