Tackling Parallelization Challenges of Kernel Network Stack for Container Overlay Networks

Jiaxin Lei[*], Kun Suo[+], Hui Lu[*], and Jia Rao[+]

[*]State University of New York (SUNY) at Binghamton
[+]University of Texas at Arlington

Abstract

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 paper, 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.

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 machines with orchestration tools, such as Apache Mesos [1], Kubernetes [12], 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 [20], Calico [2] and Docker Overlay [6]. 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 [19]). 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 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 [15]). 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 [28, 29]. 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).

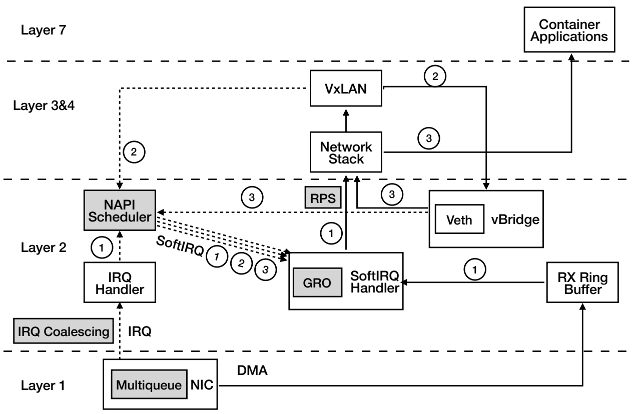

Figure 1: Illustration of data receiving path in Linux kernel.

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 paper, 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 Background & Related Work

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

Network Packet Processing. Packet processing traverses NICs, kernel space, and user space. Taking receiving a packet as an example (Figure 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 protocol stack. After being processed at various protocol layers, the packet is finally copied to the user space buffer and passed to the application.

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 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 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.

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 [21,23,27]; packet coalescing which reduces the number of packets that need to be processed by kernel network stacks (e.g., Generic Receive Offload [9] and Large Receive Offload [13]); user-space network stacks which bypass the OS kernel thus reducing context switches [10]; and data path optimizations [22,24–26].

(2) Parallelizing packet processing path: High-speed network devices can easily saturate one CPU core even with

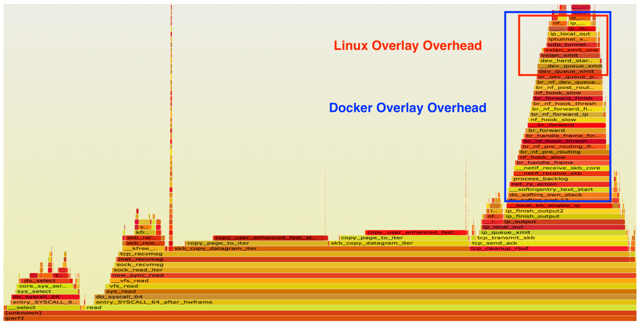

Figure 2: Function call graph along the TCP receiving path.

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 Scaling (RSS) [18]. Even without the NIC support, Receive Packet Steering (RPS) [17] 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.

3 Evaluation of Container Overlay Networks

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

3.1 Experimental Settings

We conducted experiments with three network configurations as follows:

- **The 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.
- **The 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 [14].
- **The Docker Overlay Case:** A Docker [5] 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* [3] 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* [5] on *Linux-16.04-4.4*, and used *iperf3* [11] 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.

3.2 Performance Results and Analysis

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

Figure 3 shows the TCP throughput, while Figure 5 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 4 (CPU usage breakdown for TCP) and Figure 6 (CPU usage breakdown for UDP), in the single flow case, the docker overlay consumes the same (or more) CPU usage with much less throughput, compared to the native case [1]. Figure 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 throughput keep scaling by consuming more CPU resources? Upon deeper investigation, we found that existing parallelization approaches

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

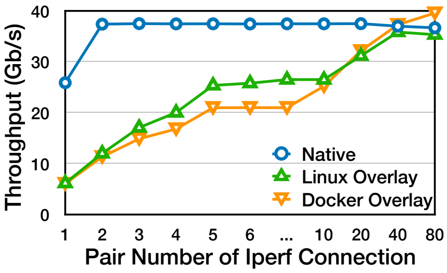

Figure 3: TCP throughput.

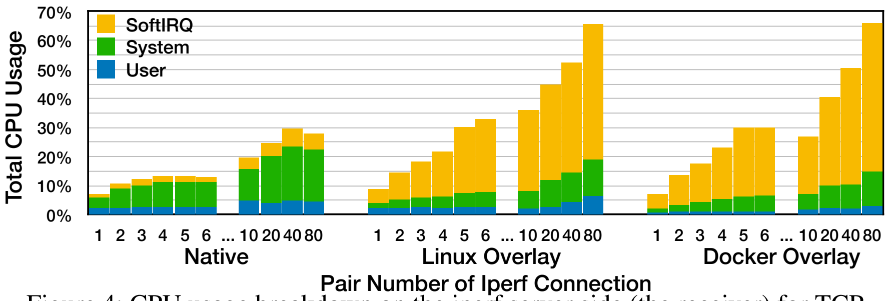

Figure 4: CPU usage breakdown on the iperf server side (the receiver) for TCP.

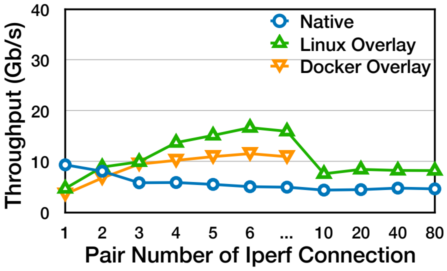

Figure 5: UDP throughput.

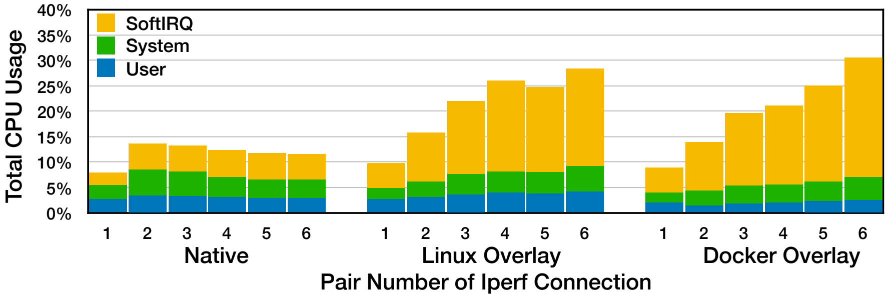

Figure 6: CPU usage breakdown on the iperf server side (the receiver) for UDP.

(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 4 and Figure 6, the CPU consumed by the kernel (i.e., the sum of the system and softirq parts) saturates one core.

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 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 4, 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 5 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.

Small Packets. It is evident that most packets in the real world have small sizes (e.g., 80% $\leq$ 600 bytes [16]). 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 7, the Docker overlay performs a bit worse with small packet sizes (64 bytes to 1

[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.

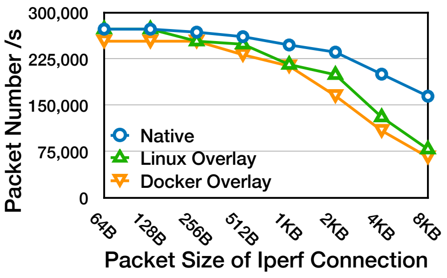

Figure 7: TCP packet processing rate.

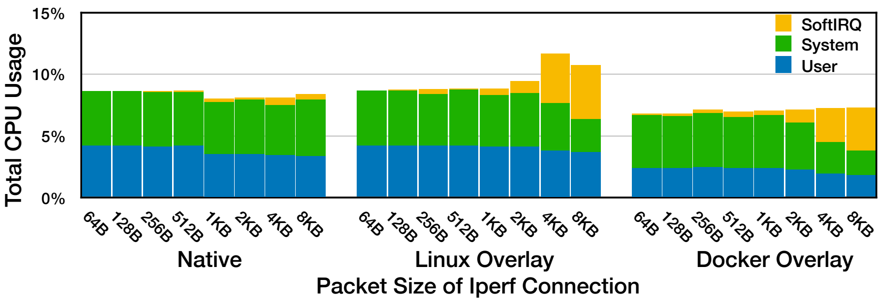

Figure 8: CPU usage breakdown on the iperf server side (the receiver) for TCP.

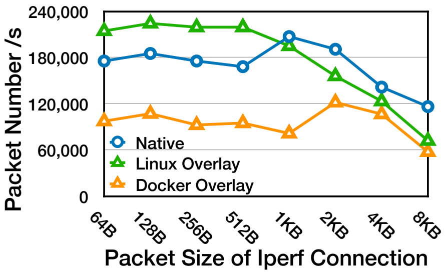

Figure 9: UDP packet processing rate.

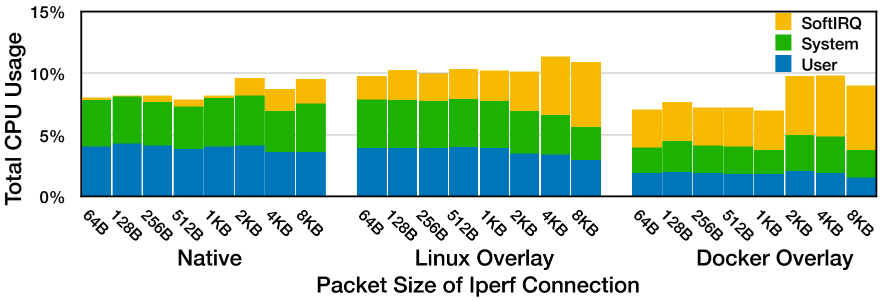

Figure 10: CPU usage breakdown on the iperf server side (the receiver) for UDP.

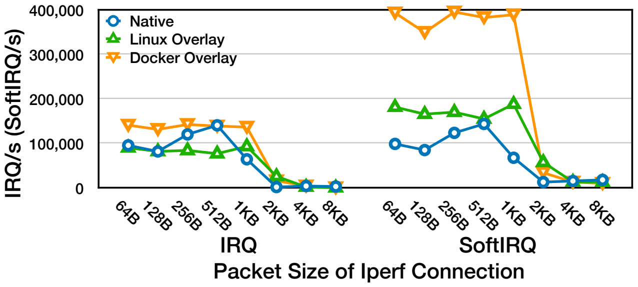

Figure 11: Interrupt number with varying packet sizes (UDP).

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 8, the Docker overlay consumes less CPU due to lower packet processing rate [3].

The more significant inefficiency is observed in the UDP case: In Figure 9, 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 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 11, ∼*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 processed 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.

4 Insights and Conclusions

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?

[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.

Discussion Topic

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.

References

[1] Apache Mesos. http://mesos.apache.org.

[2] Calico. https://www.projectcalico.org.

[3] Consul. https://www.consul.io.

[4] Containers at Google. https://cloud.google.com/containers/.

[5] Docker. https://www.docker.com.

[6] Docker Overlay Network. https://docs.docker.com/network/network-tutorial-overlay/#use-the-default-overlay-network.

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

[8] Flannel. https://github.com/coreos/flannel.

[9] Gro: Surviving 10gbp/s with cycles to spare. https://events.static.linuxfound.org/images/stories/slides/jls09/jls09_xu.pdf.

[10] Intel DPDK. https://www.dpdk.org.

[11] Iperf3. https://iperf.fr.

[12] Kubernetes. https://kubernetes.io.

[13] Large receive offload. https://lwn.net/Articles/243949/.

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

[15] Open vSwitch. https://www.openvswitch.org.

[16] Packet length distributions. http://www.caida.org/research/traffic-analysis/AIX/plen_hist/.

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

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

[19] VxLAN. https://tools.ietf.org/html/rfc7348.

[20] Weave. https://github.com/weaveworks/weave.

[21] Leopoldo Angrisani, Lorenzo Peluso, Annarita Tedesco, and Giorgio Ventre. Measurement of processing and queuing delays introduced by a software router in a single-hop network. In *Instrumentation and Measurement Technology Conference, 2005. IMTC 2005. Proceedings of the IEEE*, volume 3, pages 1797–1802. IEEE, 2005.

[22] Nathan L Binkert, Lisa R Hsu, Ali G Saidi, Ronald G Dreslinski, Andrew L Schultz, and Steven 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.

[23] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 15–28. ACM, 2009.

[24] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[25] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. 2014.

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

[27] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*, ALS '01, pages 18–18, Berkeley, CA, USA, 2001. USENIX Association.

[28] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 189–197. IEEE, 2018.

[29] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim:{OS} kernel support for a low-overhead container overlay network. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 331–344, 2019.