Accelerated Flow Processing in Kubernetes Overlay Networks

Srinath Vasudevan, Khaled Harfoush
Department of Computer Science
North Carolina State University
Raleigh, USA
e-mail: svasude5@alumni.ncsu.edu, kaharfou@ncsu.edu

***Abstract*—Kubernetes is a popular container orchestration tool which makes managing numerous containers simple through various abstractions. *Pods* are the abstraction for containers, and Kubernetes allows for pod inter-networking through the Container Network Interface (CNI) specification. Overlay networking is a technique which utilizes network tunneling across hosts to allow pods to communicate with each other using their private IP addresses. Previous work shows that overlay networking results in significant network performance degradation due to the packet encapsulation and decapsulation processing steps required for network tunneling, largely because all packet processing is usually performed on a single CPU core by default. Optimizations in the Linux kernel exist, such as Receive Packet Steering (RPS) and Receive Flow Steering (RFS), in order to parallelize packet processing per flow. The current literature does not thoroughly evaluate RFS, nor does it evaluate RPS when enabled at the container level in overlay networks. In this paper, we conduct a thorough measurement study and a performance analysis of these optimizations. Our results demonstrate the greatest improvements in average bitrate and CPU core load distribution when RFS is enabled at the host and container levels, although improvements are seen in other scenarios as well. This important work outlines actionable methods for overlay network performance gains that are applicable to public cloud systems.**

Keywords-Kubernetes; Calico; Overlay Networks; Receive Flow Steering; Receive Packet Steering.

I. INTRODUCTION

Virtual *containers* have seen increased popularity in recent years. Unlike Virtual Machines (VMs), containers virtualize the host kernel rather than the hardware. This allows containers to be smaller and have a faster creation/termination time than their VM counterparts, leading to increased portability and flexibility. Multiple technologies have been developed around containers, namely container orchestration tools, such as Kubernetes [1] and OpenShift [2], allowing for the simplified management of multiple containers in a centralized environment. A standardized specification for container networking called the Container Network Interface (CNI) [3] was created as a project by the Cloud Native Computing Foundation (CNCF) [4] in an attempt to have a standardized container networking interface. Furthermore, a wide variety of solutions were introduced to enable inter-container communication such as Calico [5], Cilium [6], and Flannel [7]. These efforts have led to the wide adoption of container orchestration tools throughout production workloads [8].

However, container orchestration tools introduce some complexity in scheduling, service routing, and communication [9], [10]. A common implementation of container networking uses a strategy called *overlay networking*, which relies on network *tunneling*, allowing the use of private IP addresses to communicate over a public network. Network tunneling encapsulates/decapsulates packets to preserve private IP addresses when transmitted over a public network. Overlay networks incur a longer data path due to this encapsulation/decapsulation necessity [9]. The elongated data path is the main contributor to the increased latencies seen in overlay networks [10]. Another problem with the longer data path is that packets are normally only processed serially on a single core, leading to a bottleneck in network throughput.

Certain technologies exist to mitigate the impact of serial packet processing. Receive Side Scaling (RSS) [11] is a hardware optimization that uses multiple receiving queues on a single NIC to accelerate packet processing. Receive Packet Steering (RPS) [12] is a software implementation of RSS that aims to distribute packet processing across many cores. Receive Flow Steering (RFS) [13] is a software extension to RPS that tries to improve RPS by processing packets on the core of the destination application buffer in order to benefit from cache locality. Both software options are supported by the Linux kernel [14] and need to be explicitly enabled on applicable network devices.

However, current research efforts do not evaluate the efficacy of RFS in a container overlay network, nor do they evaluate the efficacy of RPS and RFS when enabled at the container level [15]. Additionally, current efforts are not clear on the configurations used for RPS [15]. Overall, the benefits of these existing optimizations are inadequately studied. In this paper, we fill this gap by evaluating the benefits of introducing the RPS and the RFS optimizations in various configurations on a VXLAN-based Calico overlay network [5] using Kubernetes for container orchestration. Our work provides insights into scenarios where RPS and RFS perform the best and the inherent incompatibilities of these optimizations in overlay networks, allowing future researchers to obtain a clear idea of the current state of overlay network optimizations.

The remainder of this paper is organized as follows. In Section II, we survey related work. In Section III, we introduce basic terminology regarding Kubernetes, network tunneling, and overlay networking. In Section IV, we provide necessary background about our chosen overlay network, RPS, and RFS. In Section V, we evaluate the performance of the RPS and RFS optimizations. We finally conclude in Section VI.

II. Related Work

Many technologies exist to support communication between hosts including Network Address Translation (NAT), host networking, routing via BGP, and overlay networks [10]. Among these, overlay networks are very common especially in container orchestration environments due to their flexibility and straightforward deployment.

Studies were conducted to compare the performance of these technologies [10] [16]. In [10], Suo et al. analyzed various methods for intra and inter-host networking among containers under various protocols and network conditions. Among the overlay networks analyzed were Calico [5], Weave [17], Flannel [7], and Docker Overlay [18]. Compared to other networking options, overlay networks performed worse in terms of throughput, latency, and network launch time. However, comparing overlay network performance to technologies, such as NAT or host mode networking is an unfair comparison as they are built for different purposes. Overlay networking provides an easy-to-use networking setup among containers, which is easier to manage in changing network topologies. In [9], Suo et al. studied the reasons for the network overhead in overlay networks. They observed high amounts of scheduled hardware and software interrupts in overlay networks compared to regular host networking. This increase coupled with the fact that multicore systems are unable to effectively parallelize the excessive software interrupts without specialized hardware or protocols leads to an imbalance of compute power in such systems. This is largely attributed to the longer data path of a packet due to the added overhead from encapsulation/decapsulation and the container bridge network. In [19], Lei et al. highlighted that RPS and RFS are both flow-level parallelization methodologies, and suggested that packet-level parallelization would increase performance for single-flow situations. The authors also found that multiple container flows do not saturate a particular network link, indicating that overlay networks do not scale well, likely due to the large amount of context switches. This phenomenon is further exacerbated with small packet sizes.

Various efforts were aimed at optimizing overlay network performance, whether offering software solutions or hardware solutions. On the **software** solutions front, in [20], Lin et al. propose SlimFast, which uses a host machine's socket directly from the container to eliminate encapsulation overhead while preserving the ability to utilize a container's private IP address without knowledge of the underlying host IP. However, SlimFast intercepts system calls related to socket communication, which requires explicit kernel support. In [15], Lei et al. propose mFlow, a technology implementing packet-level parallelism [19] for *stateless* steps in the networking stack. These include steps where in-order processing is unnecessary. For steps that require in-order processing (such as TCP or L7 protocols), the packets are reassembled in order before that step is processed to preserve order. Instead of processing an entire flow in parallel to other flows, mFlow breaks up a single flow into subflows and processes the subflows in parallel. Naturally, the subflows retain their ordering. mFlow improves TCP and UDP throughput when compared to a regular overlay networks. Performance increases are also consistently seen when compared to an RPS enabled overlay network. Proposed solutions do not evaluate the efficacy of RFS and all include technologies that need explicit kernel support. RPS and RFS are already supported by the Linux kernel [14], so they can readily be used in public cloud environments - an effort which we undertake in this paper.

On the **hardware** solutions front, hardware-acceleration is an option. In [21], Ma et al. propose a SmartNIC-based software-hardware design to enhance the performance in a cloud environment. The main idea is to offload the encapsulation/decapsulation operations typically done by the kernel to the SmartNIC. While this proved to reduce CPU overhead while lowering latency and increasing throughput [21], the results show that throughput does not scale as well when the core count becomes high (greater than around 4-6 cores). This is attributed to the fact that SmartNIC does not distribute packets effectively to the different cores. Note that hardware offloading for container networks, as opposed to software solutions, leads to security concerns due to multi-tenancy. Finally, the adoption of SmartNIC hardware in a public cloud environment is challenging. Software solutions are readily supported by the Linux kernel, so any potential benefits can be readily adopted.

Overall, the issue of overlay network optimization has many solutions being researched. However, the existing optimizations available in the Linux kernel have not been exhaustively and adequately studied in this context. In this work, we systematically study RPS and RFS in various configurations to cover the gaps in previous work.

III. Terminology

In the following subsections, we introduce necessary terminology for understanding Kubernetes and overlay networking concepts.

A. Kubernetes

Kubernetes is a container orchestration framework that simplifies many management tasks of multi-container environments [1]. Kubernetes has many abstractions and various terms we use throughout this paper. These are defined as follows. A *Cluster* is a logically isolated Kubernetes environment, consisting of one or more host machines; a.k.a. *nodes*. Nodes are able to communicate with each other to match the desired state of the cluster. Nodes are typically implemented as Virtual Machines. A *Pod* is the smallest Kubernetes abstraction, typically used to represent a single container. A pod can also contain multiple containers, but this is mainly common for monitoring purposes. In this paper, Kubernetes pods are synonymous to a single container. A *Service* is a networking abstraction which enables a set of pods to be accessible by a static IP address. This static IP routes to a set of pods automatically, and a DNS name for the service is also created. In this paper, each service we create

will only route to a single pod. *CNI Plugins* [3] enable inter-pod networking in Kubernetes. These do not come installed by default on Kubernetes and must be explicitly installed. Many different implementations exist. In our experiments, we use Calico [5].

B. Network Tunneling

Network tunneling [22] is a technology that allows for a packet of some protocol to be transmitted over a network by encapsulating it in the header of another protocol. This is useful in the event the network does not support the encapsulated protocol. At the destination, the packet is decapsulated and the original packet is retrieved for processing. The destination should be able to process the original packet and its protocol. Popular tunneling protocols include VXLAN [23], GRE [24], and IPIP [25].

C. Container Overlay Networks

In an overlay network, a virtual Layer 2 network device (such as a Linux bridge) is created to handle communication between containers. The containers' virtual devices are connected via a virtual ethernet (`veth`) pair. Overlay networks take advantage of network tunneling to allow pods on separate host machines to communicate using their private IPs without any explicit knowledge of the destination host's IP. Another network interface is used to perform encapsulation and decapsulation of packets. The encapsulated packet header contains the IP of the destination host which the destination pod resides in. This destination host IP is typically referenced through a distributed KV store, such as `etcd` [26].

In networking contexts, two types of interrupts are invoked: software interrupt requests (`softirqs`) and hardware interrupt requests (`hardirqs`). `hardirqs` are always invoked by hardware events, such as a packet arriving at a NIC. This triggers a hardware interrupt handler to handle the packet and place it in the proper buffer. A `softirqs` is invoked to actually process the packet once it is available to the kernel. Most network processing occurs as a result of `softirqs`, and they take on the majority of packet processing workloads.

In a receive path, a packet first traverses the host network stack once the NIC copies the packet into a buffer and raises the first `softirq` [9]. After the software interrupt handler is invoked, processing continues up to Layer 3 and layer 4. This is where the packet is decapsulated and put in a queue for the virtual Layer 2 device. Another `softirq` is raised to handle this inner packet, which is emitted to the container through the veth pair from the virtual Layer 2 device. This in turn invokes yet another `softirq` to process the packet in the container's network stack.

IV. Background

In the following subsections, we give essential background on our chosen overlay network, overlay network processing and existing software-based network processing optimizations.

A. Calico

We use Calico [5] as our overlay network. Calico creates a new network interface (virtual Layer 2 device) in the host network namespace named `cali*` to communicate with a pod. Each pod has its own network namespace and is able to communicate with `cali*` via a virtual ethernet (veth) pair. The network interface for this pair on the pod is `eth0`. The backend installed by Calico depends on the type of encapsulation used. We use Calico configured with VXLAN [23] encapsulation, so another network interface in the host network namespace named `vxlan.calico` is created. This is where encapsulation and decapsulation according to the VXLAN tunneling protocol occurs. When installing Calico with Kubernetes, there is no need to set up a separate datastore (such as `etcd`). There is an option to directly interface with the Kubernetes API server to find pod-to-host IP mappings, which makes management easier. We use this option in our experiments. Figure 1 shows the general data path of a pod (pod 1 on host A) sending a packet to another pod on another host (pod 2 on host B), as well as all the network devices it traverses.

B. Packet Processing Overhead

A received packet needs to traverse the entire host networking stack as well as the entire container networking stack, incurring significant overhead. Encapsulation/Decapsulation and the invocation of corresponding `softirqs` are the main contributors to the overhead. Up to 3x more software interrupt requests (`softirqs`) compared to a native host network receive path were observed in [9].

The main reason behind the overhead from increased `softirqs` comes from how Linux schedules `softirqs` onto cores. Typically, hardware interrupt requests (`hardirqs`) are processed on a single chosen core [9]. Once a NIC receives a packet, a `hardirq` is raised and processed on that particular core. Any `softirqs` that result from the `hardirq` are typically processed on the same core the `hardirq` was processed on. This means that the first `softirq` (and every subsequent one) are processed on the same core. Furthermore, for a single queue NIC, all `hardirqs` and `softirqs` must be processed serially on that single core, mostly to avoid out-of-order packet processing. In a multicore system this can easily exhaust a single core's resources while leaving other cores idle.

C. Receive Packet Steering (RPS)

Receive Side Scaling (RSS) [11] is a hardware optimization that allows `softirq` processing to be distributed among multiple cores. RSS works by having multiple input queues (usually bound by the number of cores) for a single NIC, which means that the content of each queue can be processed by a different core. Receive Packet Steering (RPS) [12] is the software implementation of RSS. Rather than requiring multiple receive queues, RPS creates a hash based on relevant Layer 3 and Layer 4 information (IP and port) to choose an arbitrary CPU to process the packet on. This is an example of flow-level parallelism, where packets of the same flow get

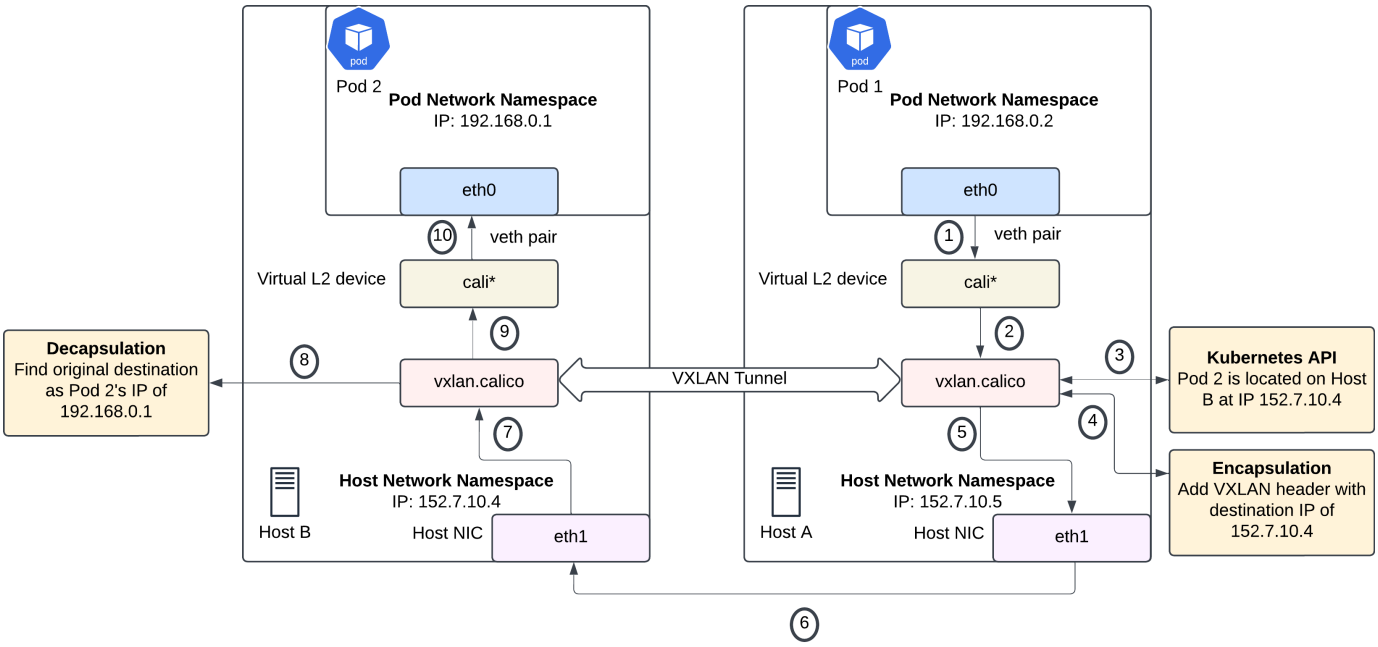

Figure 1. An example of how the Calico overlay network with VXLAN encapsulation transmits and receives packets. Pod 1 on Host A is sending a packet to Pod 2 on Host B.

processed on the same CPU. Figure 2 shows a simplified flow of a packet in a receive path with RPS enabled.

RPS must be explicitly enabled on a per-interface level. In order to do so, the file `/sys/class/net/{DEVICE}/queues/rx-{QUEUE-NUMBER}/rps_cpus` must be modified. In order to enable RPS on all cores of the eth0 device on its only receive queue, we write a value of `f` in the file `/sys/class/net/eth0/queues/rx-0/rps_cpus`.

D. Receive Flow Steering (RFS)

RFS [13] is an extension of RPS that aims to process a packet on the same CPU that the destination application runs on to take advantage of cache locality - although this is not guaranteed to happen. In order to achieve this goal, RFS uses one of the RPS generated hash tables and tracks the last used CPU for processing the flow. In order to enable RFS, we modify two files: `/proc/sys/net/core/rps_sock_flow_entries` and `/sys/class/net/{DEVICE}/queues/rx-{QUEUE-NUMBER}/rps_flow_cnt`. The former is typically set to 32768 to indicate the maximum number of connections. The latter indicates the number of expected flows per device queue. In a single queue device, this can be the same value as the `rps_sock_flow_entries`.

It should be noted that both RPS and RFS are processing techniques that exploit *flow parallelization*. Since the hash is based on Layer 3 and Layer 4 packet information, all processing for a single network flow is done on the same core. This is the easiest way to implement network processing parallelization as it avoids out-of-order packets. As a result of flow-level parallelization, RPS and RFS are not designed to increase performance in single-flow scenarios. Some techniques, such

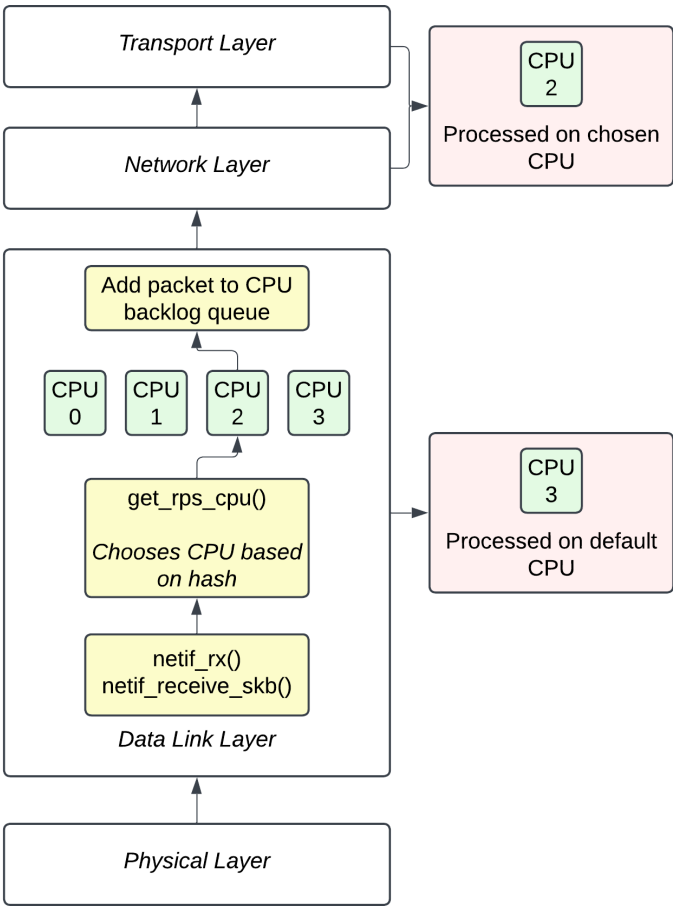

Figure 2. Simplified flow of a network receive path with RPS enabled on 4 cores in the Linux networking stack. The default core for packet processing is assumed to be CPU 3.

as those proposed in [15] and [27], implement packet-level parallelism with methods to overcome out-of-order processing challenges.

V. Performance Evaluation

In the following subsections, we describe our experimental setup, evaluation tools, experiment details and the results.

A. Experimental Setup

The Kubernetes cluster we use consists of three nodes, where each node is a VM created with KVM as the hypervisor. One of these nodes is a control plane node, and the other two are exclusively used for the experiments (worker nodes) – Refer to Figure 3. Each VM is on a separate Dell PowerEdge R930 machine and is running Ubuntu 22.04 LTS. All VMs are provisioned with 4 cores and 8 GB of RAM. Each physical machine has a bonded network link to two top of rack switches. All machines we use are connected to the same two switches. We use `kubeadm` to install Kubernetes version 1.31.1 and use `containerd` as our container runtime. We installed the Calico CNI as our overlay network in a VXLAN encapsulation mode. All the code for the experiments can be found at this github repository [28].

In our experiments, we introduce different number of flows between the two worker nodes. A single flow between two worker nodes is a single client/server connection where the client and server continuously send packets to each other throughout the experiment duration. All clients are placed on one node and all servers are placed on the other node. This ensures that traffic flows from one VM to a separate VM. The number of clients and server pairs can be adjusted variably - we test number of pairs increasing by a power of 2 until we reach 16 client/server pairs. In this paper, we use "replicas" to mean the number of client/server pairs. For instance, 2 replicas means two clients and two servers in total.

B. Performance Metrics

For each experiment, we report on the following performance metrics: (1) The *normalized average bitrate*, (2) the *CPU Idle Percentage*, and (3) *CPU `softirqs` Percentage*. All our experiments are repeated to plot averages and 95% confidence intervals. The average bitrate in bits/sec for the duration of each experiment is normalized relative to the baseline experiment bitrate (when no RPS or RFS optimizations are deployed). A value larger than 1 reflects an improvement in bitrate over the baseline case. We rely on iPerf3, a network performance tool, to measure bitrates [29]. In our experiments, we set up an iPerf3 server on one pod and an iPerf3 client on another. From the client, we reference the server via its Kubernetes service's DNS name and let the client repeatedly send an array of 128 KB for 30 seconds over TCP. We then average the bitrate measured at each server instance. We use the mpstat [30] tool to collect CPU usage percentages by each core, and report on idle CPU percentages and the percentage of time a CPU is used to process `softirqs`. We run mpstat on the server node and use the `-P ALL` flag to report information for each core every second in each experiment, then compute averages.

C. Experiments

The overlay network configurations that we evaluate are: (1) **Baseline** (no RPS or RFS optimizations), (2) **RPS** (RPS enabled on all cores), (3) **RPS+** (RPS enabled on all cores and container-level RPS enabled), (4) **RFS** (RFS enabled), and (5) **RFS+** (RFS enabled and container-level RFS enabled). We do not optimize the baseline scenario in any unique way compared to the other scenarios.

We only enable RFS or RPS on the iPerf3 server's node exclusively on the following three network interfaces: (1) `vxlan.calico` interface: The VXLAN backend, (2) `cali*` interface: The Layer 2 switch to communicate with pods, and (3) `eth1` interface: The virtual NIC of the host machine. We keep these optimizations disabled on all other interfaces on the server node. For the RPS+ and RFS+ experiments, we enable the respective optimization at the container level in addition to the host interfaces. For example, RFS+ enables RFS at the three previously mentioned network interfaces as well as all interfaces in the container. This requires us to remount the `/sys` filesystem as read-write (containers have the `/sys` filesystem as read-only by default). In Kubernetes, the pod is given root and privileged access to perform this operation.

For the options with RPS enabled on all cores, a value of `f` is written to the `rps_cpus` file for all network interfaces for which we wish to enable RPS on. For options that enable RFS, the `rps_sock_flow_entries` file and `rps_flow_cnt` files for each network device are set to 32768 as described in Section IV.

D. Results

Figures 4 (a) and (b) plot the CPU idle time percentages for all cores when using 1 replica and 16 replicas, respectively. The figure reveals that a single replica does not stress the CPU and leads to more than 90% CPU idle time for all cores, while 16 replicas stress the cores leading to around 60% of idle time for cores 0, 1 and 2, and leading to only about 30% of idle time for core 3. As a result, single replica scenarios are not expected to benefit significantly from flow parallelization optimizations. This is largely due to the fact RPS and RFS perform much better with multiple flows since they rely on flow-level parallelization. Note that packet processing is done on Core 3 by default, evidenced by the very low idle core percentage for the baseline case. This can be further evidenced by inspecting Figures 5 (a) and (b), which plot the CPU percentage of time spent handling software interrupts (`softirqs`) for all cores when using 1 replica and 16 replicas, respectively. Focusing on the 16 replicas case, core 3 is handling more (`softirqs`) than the other cores. Also, note in Figure 5 (b), that all optimizations reduce the percentage of time that core 3 spends servicing `softirqs` compared to the baseline case. Among these optimizations, RFS+ offers the most reduction, and thus the best load balancing among the cores, followed by RFS and RPS+.

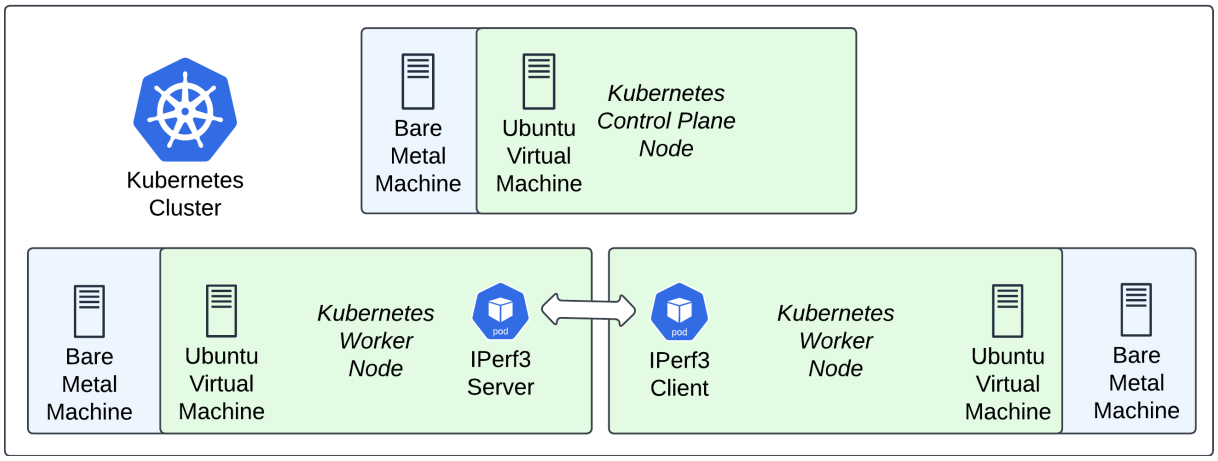

Figure 3. The general experiment setup in the Kubernetes environment.

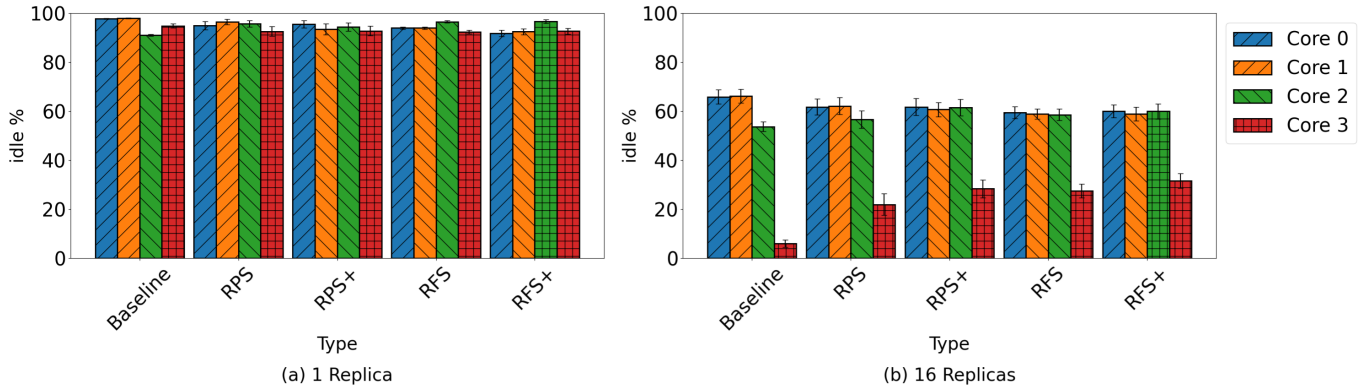

Figure 4. Average idle CPU percentage by cores for each optimization with (a) a single replica and (b) 16 replicas.

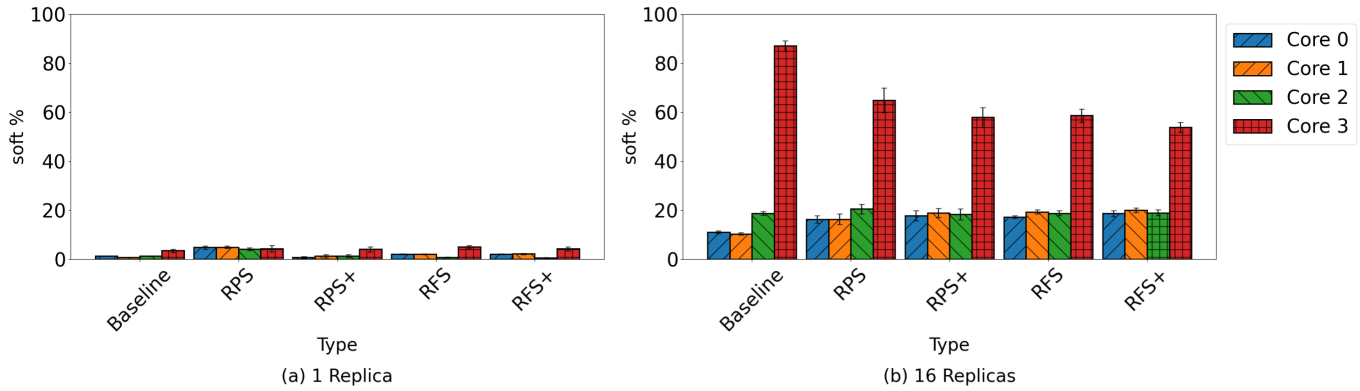

Figure 5. Average CPU percentage used for software interrupts by cores for each optimization with (a) a single replica and (b) 16 replicas.

The reason we do not see even more balanced numbers in terms of core utilization is due to how RPS (and in turn, RFS) protocols work. The generated hash that determines the core to process a packet on relies on Layer 3/Layer 4 information, meaning that only the protocol processing steps (Layer 3/Layer 4) can be parallelized [12]. In an overlay network, the final packet processing step at the destination pod and the VXLAN decapsulation processing step are the only two steps that will process packets at the protocol level. All processing up to Layer 2, including the transmission of packets from the Layer 2 bridge to actual pods, is not steered to any other cores. The added step to transmit packets from the virtual Layer 2 bridge to the pods leads overlay networks to have a disproportionate amount of non-protocol processing - likely reducing the efficacy of RPS and RFS optimizations. This indicates that overlay networks may not be very compatible with existing flow parallelization methods.

Figure 6 shows the normalized average bitrates across the different optimizations and replica counts. One can make the following observations: (1) The 95% confidence interval shrinks as the number of replicas increases. (2) The normalized bitrate resulting from RPS, RPS+, RFS, RFS+ dips going from 1 replica to 2 replicas, then improves as the number of replicas increases. Having only 2 replicas does not stress the server's CPU enough to see significantly high performance benefits from parallelized flows, therefore possibly exacerbating the slight software overhead that RPS incurs, leading to slightly decreased performance. (3) RPS and RPS+ optimizations improve the average bitrate over the baseline setup, except in the case of 2 replicas; while RFS and RFS+ improve the average bitrate over the baseline setup in all scenarios. (4) RPS performs better than RPS+ for the 1 and 2 replicas cases, while RPS+ performs better than RPS when using more replicas. The same observation holds for RFS performance compare with RFS+. (5) RFS+ leads to the best normalized average bitrate when the number of replicas is more than or equal to 4 (the number of cores), while RFS leads to the best normalized average bitrate when the number of replicas is less than the number of cores. Even though RFS is based on RPS, the extra cache locality benefits are the reason for the increased performance.

VI. Conclusion and Future Work

In a VXLAN-based overlay network, our work demonstrates that the largest bitrate improvements are seen when Receive Flow Steering (RFS) is enabled at the host and container levels, reaching an average increase of up to 24% over the baseline case. Enabling these optimizations on only the host shows marginal improvements in bitrate around $8-9\%$ over the baseline. Generally, more replicas provide greater benefits from RPS and RFS optimizations. However, due to overlay networks having a disproportionate amount of non-protocol level network processing, RPS and RFS are not able to exploit flow parallelism to its full potential. These findings show that for high throughput applications with frequent inter-host communication patterns, enabling RFS or RPS at the host and container levels could improve performance.

In order to enable these optimizations at the container-level, privileged container access is required in order to remount the `/sys` filesystem with read-write permissions, since containers have this filesystem as read-only by default. This introduces security implications if an adversary is able to gain access to the container for example. We intend to investigate these as a future endeavor.

Furthermore, we plan on extending our research along different fronts, such as (1) exploring how RPS and RFS optimizations perform for UDP workloads since RPS and RFS compute hashes differently based on the protocol used, which could affect the behavior of how packets are distributed to different cores; (2) studying the efficacy of flow-parallelization optimizations when using different encapsulation protocols; (3) evaluating optimizations in CNIs that do not rely on overlay networking; (4) examining the power tradeoff of these optimizations against throughput gains; (5) evaluating tail latency as a result of these steering techniques.

References

[1] "Kubernetes Documentation," Kubernetes, 20-Apr-2024. [Online]. Available from: https://kubernetes.io/docs/home/ 2026.03.08

[2] "OpenShift Container Platform 4.21", Red Hat Documentation. [Online]. Available from: https://docs.redhat.com/en/documentation/openshift_container_platform/4.21 2026.03.08

[3] CNI. [Online]. Available from: https://www.cni.dev/ 2026.03.08

[4] C. Study, "Cloud Native Computing Foundation," CNCF, 15-Nov-2024. [Online]. Available from: https://www.cncf.io/ 2026.03.08

[5] "About Calico," Calico Documentation. [Online]. Available from: https://docs.tigera.io/calico/latest/about/ 2026.03.08

[6] "Cloud Native, eBPF-based Networking, Observability, and Security," Cilium. [Online]. Available from: https://cilium.io/ 2026.03.08

[7] Flannel-Io, "flannel," GitHub. [Online]. Available from: https://github.com/flannel-io/flannel 2026.03.08

[8] J. Pelletier, "2024 Kubernetes Benchmark Report: The Latest Analysis of Kubernetes Workloads," CNCF, 26-Jan-2024. [Online]. Available from: https://www.cncf.io/blog/2024/01/26/2024-kubernetes-benchmark-report-the-latest-analysis-of-kubernetes-workloads/ 2026.03.08

[9] K. Suo, Y. Shi, A. Lee, and S. Baidya, "Characterizing networking performance and interrupt overhead of container overlay networks," Proceedings of the 2021 ACM Southeast Conference, pp. 93–99, Apr. 2021.

[10] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of Container Networks," IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, Apr. 2018.

[11] "8.6. receive-side scaling (RSS)" 8.6. Receive-Side Scaling (RSS) — Red Hat Product Documentation. [Online]. Available from: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rss 2026.03.08

[12] J. Corbet, "Receive packet steering," Receive packet steering [LWN.net], 17-Nov-2009. [Online]. Available from: https://lwn.net/Articles/362339/ 2026.03.08

[13] J. Edge, "Receive flow steering," Receive flow steering [LWN.net], 07-Apr-2010. [Online]. Available from: https://lwn.net/Articles/382428/ 2026.03.08

[14] "The linux kernel," Scaling in the Linux Networking Stack - The Linux Kernel documentation. [Online]. Available from: https://docs.kernel.org/networking/scaling.html 2026.03.08

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

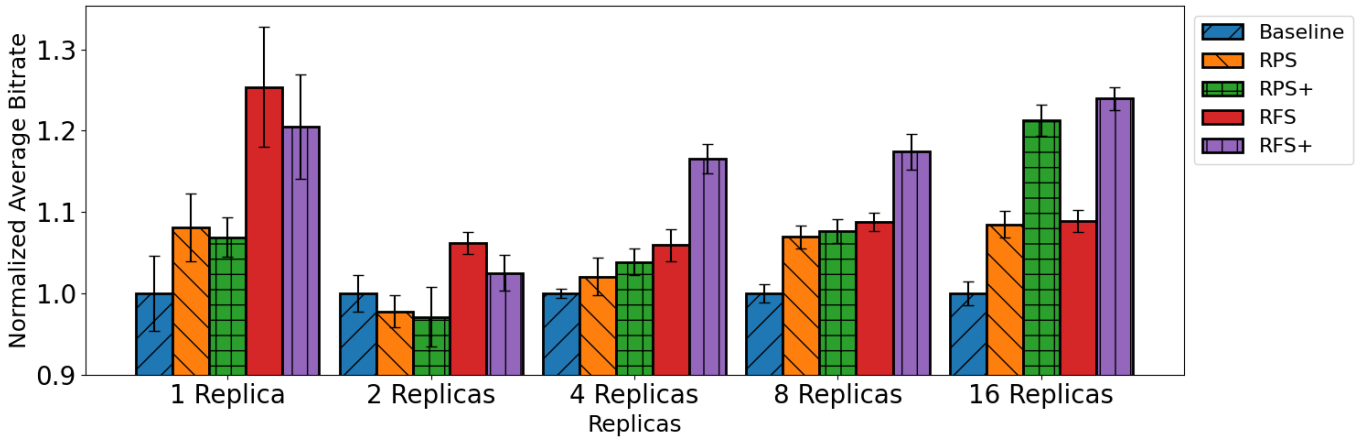

Figure 6. Normalized average bitrate for different number of replicas.

[16] S. Novianti and A. Basuki, "The performance analysis of Container Networking Interface Plugins in Kubernetes," 6th International Conference on Sustainable Information Engineering and Technology 2021, vol. 9, pp. 231–234, Sep. 2021.
[17] Weaveworks, "Weaveworks/weave," GitHub. [Online]. Available from: https://github.com/weaveworks/weave 2026.03.08
[18] "Overlay Network Driver," Docker Documentation. [Online]. Available from: https://docs.docker.com/engine/network/drivers/overlay/ 2026.03.08
[19] J. Lei, K. Suo, H. Lu, and J. Rao, "Tackling parallelization challenges of kernel network stack for Container Overlay Networks," USENIX, 01-Jan-1970. [Online]. Available from: https://www.usenix.org/conference/hotcloud19/presentation/lei. 2026.03.08
[20] F. Lin, X. Zhang, G. Chen, L. Chen, K. Li, and H. Jiang, "Slim and fast: Low-overhead container overlay network with fast connection setup," IEEE Transactions on Cloud Computing, vol. 12, no. 1, pp. 1–12, Jan. 2024.
[21] Y. Ma, S. Smith, B. Dai, H. Franke, B. Sukhwani, S. Asaad, J. Xiong, V. Kindratenko, and D. Chen, "UNINET: Accelerating the Container Network Data Plane in IaaS clouds," 2024 IEEE 17th International Conference on Cloud Computing (CLOUD), vol. 33, pp. 115–127, Jul. 2024.
[22] "What is Tunneling?," Cloudflare. [Online]. Available from: https://www.cloudflare.com/learning/network-layer/what-is-tunneling/. 2026.03.08
[23] "What is VXLAN?," Juniper Networks. [Online]. Available from: https://www.juniper.net/us/en/research-topics/what-is-vxlan.html 2026.03.08
[24] "What is GRE Tunneling?," Cloudflare. [Online]. Available from: https://www.cloudflare.com/learning/network-layer/what-is-gre-tunneling/ 2026.03.08
[25] "5.2. IP in IP Tunneling," IP in IP Tunneling. [Online]. Available from: https://tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.tunnel.ip-ip.html 2026.03.08
[26] "etcd," etcd, 22-Mar-2024. [Online]. Available from: https://etcd.io/ 2026.03.08
[27] J. Lei, M. Munikar, K. Suo, H. Lu, and J. Rao, "Parallelizing packet processing in container overlay networks," Proceedings of the Sixteenth European Conference on Computer Systems, Apr. 2021.
[28] S. Vasudevan, "srinva/accel-overlay-NW," GitHub. [Online]. Available from: https://github.com/srinva/accel-overlay-nw 2026.03.07
[29] "IPerf," iPerf.fr. [Online]. Available from: https://iperf.fr/iperf-doc.php 2026.03.08
[30] "MPSTAT(1): Report processors related statistics," Linux Man Page. [Online]. Available from: https://linux.die.net/man/1/mpstat 2026.03.08