Accelerated Flow Processing in Kubernetes Overlay Networks

North Carolina State University
Srinath Vasudevan, Dr. Khaled Harfoush
svasude5@alumni.ncsu.edu, kaharfou@ncsu.edu

Cloud Computing 2026

Introduction

Kubernetes Overview:

- Manages containers across a **cluster** of machines (nodes).
- A **Pod** is the smallest unit, usually containing one or more containers.
- **Nodes** are worker machines where **Pods** run, managed by the Kubernetes master.

Kubernetes Networking:

- Kubernetes uses **Container Network Interface (CNI)** plugins enable networking.
- Various implementations exist, including **routing based** and **NAT** implementations.
- Overlay networks (e.g., VXLAN) are common but face **latency** and **encapsulation overhead** issues.
- Overlay networks are **low-maintenance** and **scalable**.

Need for Improvement:

- Optimized overlay networks are essential to reduce bottlenecks in **multi-host environments**.

Overlay Networks

Implementation Details

- Overlay networks rely on **encapsulation** and **decapsulation** for inter-host communication.
- Pods communicate via private IP addresses.
- Virtual ethernet pairs link containers to virtual network.
- Virtual L2 device connects containers across hosts via **network tunneling**.

Challenges of Overlay Networks

- Introduces multiple **software interrupt requests (softirqs)**.
  - Up to 3x more than typical host networking
- All these softirqs are typically processed on a single core.
- Software interrupt imbalance leads to poor multi-core utilization.

Motivation

Growing Adoption of Containerized Applications

- Need for performant and scalable networking solutions in multi-host container orchestration platforms.

Overlay Networks Widely Used but Inefficient

- High network overhead due to encapsulation and long data path.
- Current comparisons with NAT and host networking are unfair as these are not designed for highly dynamic environments.

Objective

- Explore methods to improve performance of overlay networks in Kubernetes
- Examine resource usage to hypothesize about scalability of such methods

Current Technologies

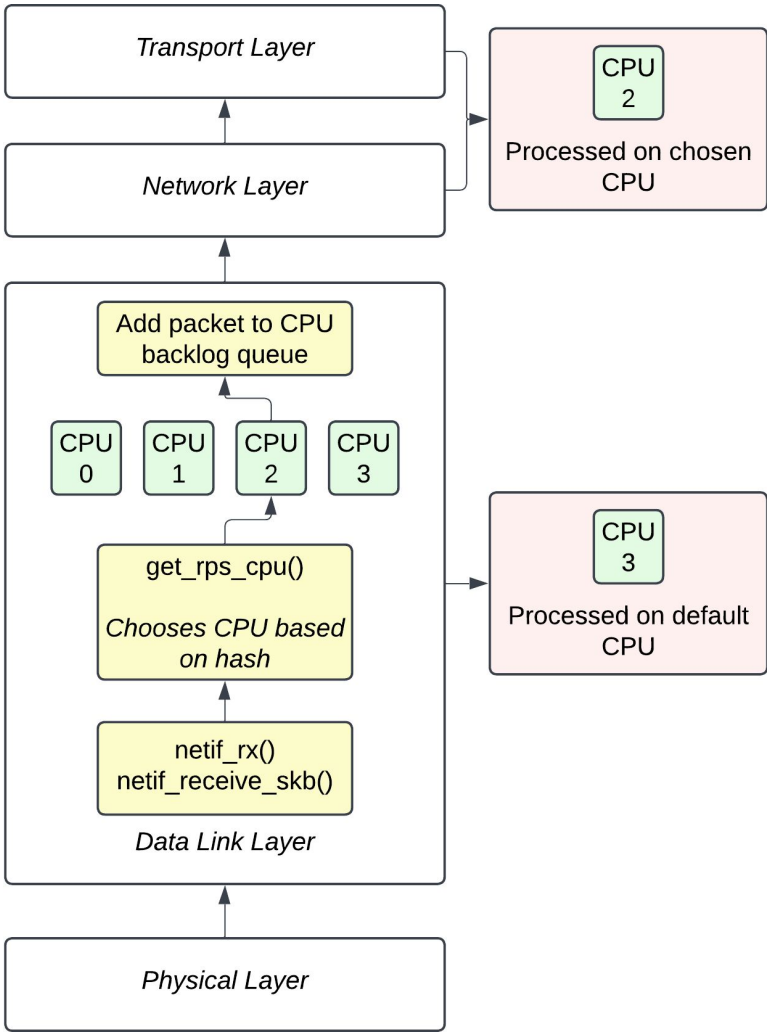
Receive Packet Steering

- Used to steer packets to certain cores to be processed
- Computes a hash based on IP and port to decide which core to use
- Need to set the usable processors in rps_cpus
- Does not need hardware support

Receive Flow Steering

- Extension of RPS
- Aims to take advantage of CPU cache locality
- Forwards packets based on the location of the application
- Need to specify maximum number of expected concurrent connections
- Need to specify number of flow entries per receive queue

RPS Flow Overview



Current Literature

Gaps in Literature

- Does not consider RFS as an option
- One study shows that RPS is not very effective at increasing throughput
  - Although, some performance increases seen in TCP flows
  - Doesn't mention RPS configuration
  - Only enable RPS at overlay and host level

Proposed Solutions

- Proposes alternative technologies for overlay networks
  - Slim and Fast
    - Utilize host socket directly
  - mFlow
    - Packet-level parallelism rather than flow-level
  - UniNet
    - Hardware-offloading decapsulation support
- RPS and RFS already supported by Linux Kernel

Our Contribution

Contributions

- Evaluate overlay network performance using Receive Packet Steering (RPS) and Receive Flow Steering (RFS) in different configurations
  a. Evaluate various flow configurations
  b. Enable RPS/RFS at the host & overlay level
     i. Also enable at the container level
  c. Provide readily available performance optimization options

Methodology

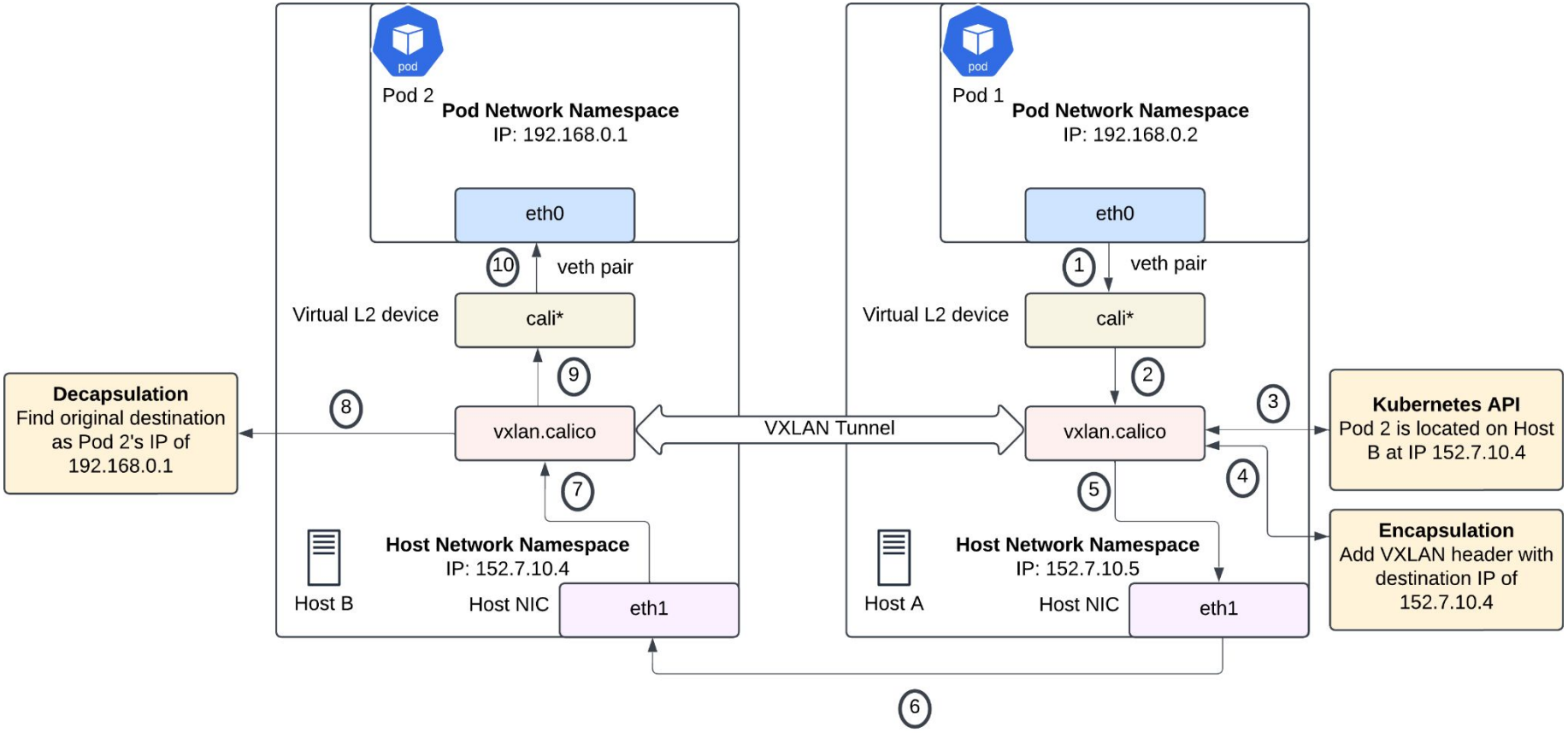
- Utilize iPerf3 to evaluate network performance
- Observe interrupt balance among cores for each competitor
- Measure CPU utilization by core for each competitor
- Draw conclusions from data on any performance benefits

Experimental Variables

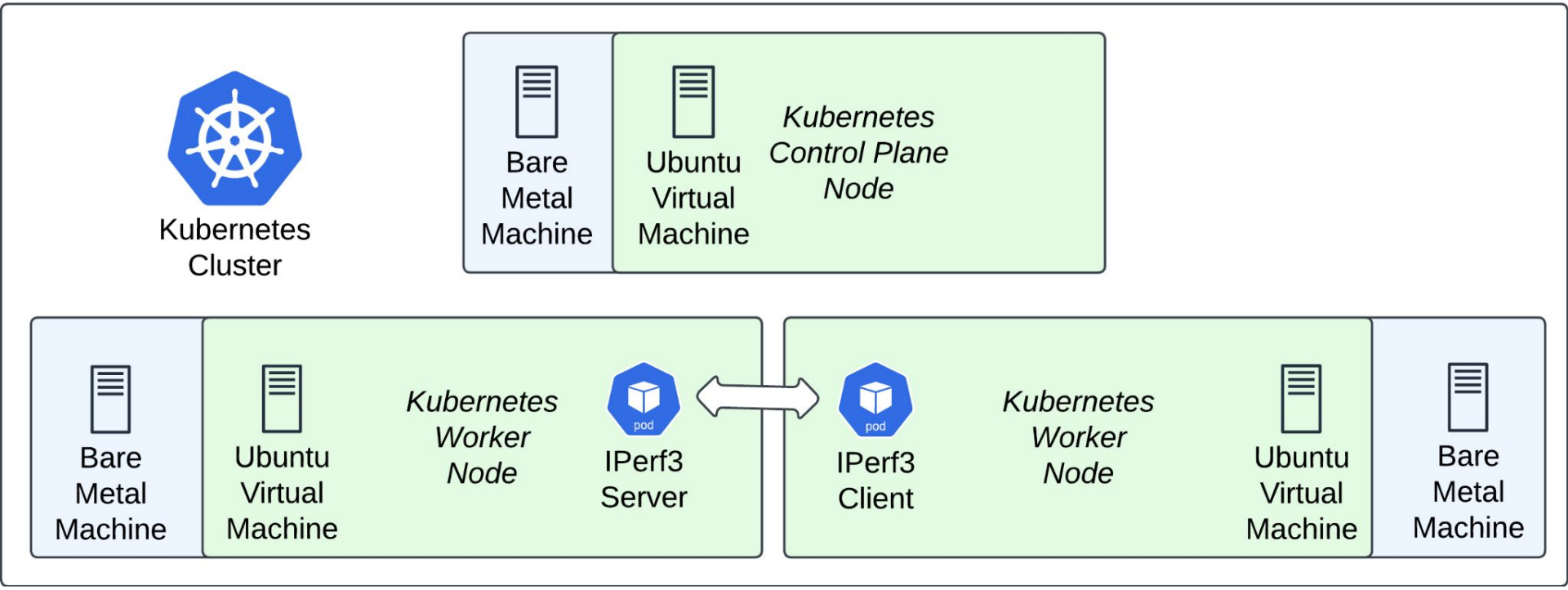
- Use Calico CNI as plugin in the following general configurations:
  - **Baseline:** VXLAN
  - **RPS**: VXLAN + RPS (all cores)
  - **RPS+**: VXLAN+ RPS (all cores) + container RPS
  - **RFS**: VXLAN + RFS
  - **RFS+**: VXLAN + RFS + container RFS
- Test all scenarios with 1, 2, 4, 8, and 16 client-server pairs
  - 4 cores per Kubernetes node
- Test average bitrate across server instances
  - Using iPerf3
  - Clients send 128 KB over 30 seconds
  - Servers scheduled on single node separate from clients
- Test CPU core utilization among all configurations
  - mpstat
  - Measure cpu idle %
  - Measure cpu softirq %

Calico Overlay Network

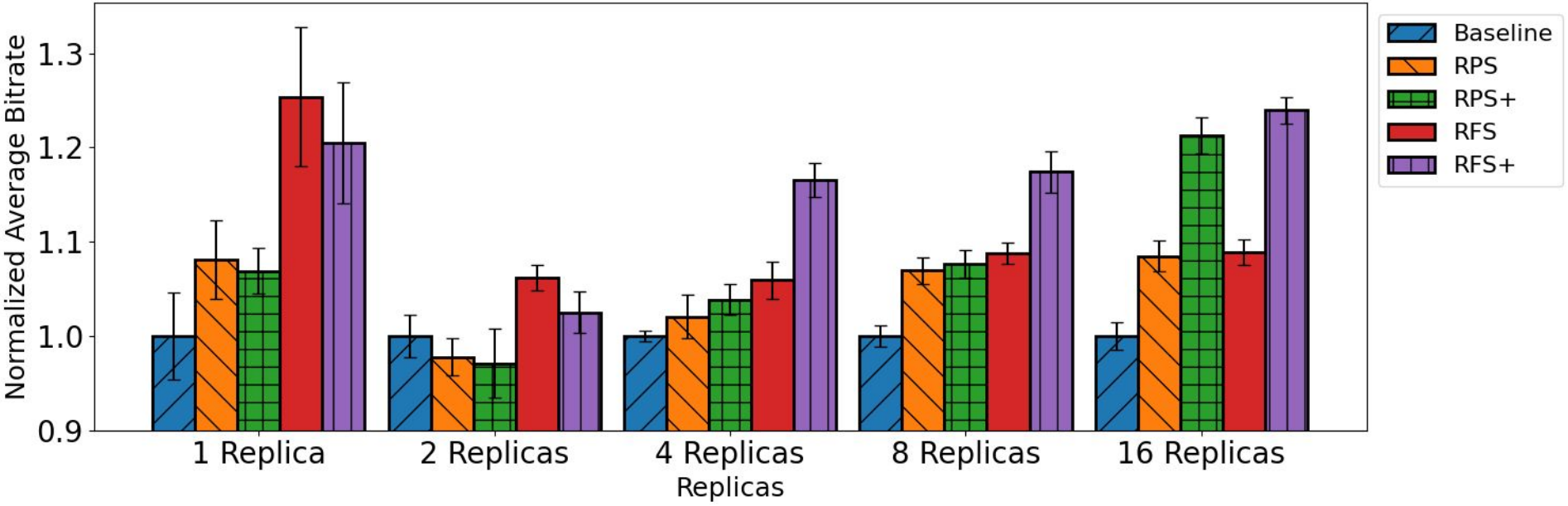
- Two additional network devices
  - cali* interface: Virtual L2 device used to communicate with pods
  - vxlan.calico interface: Performs encapsulation/decapsulation



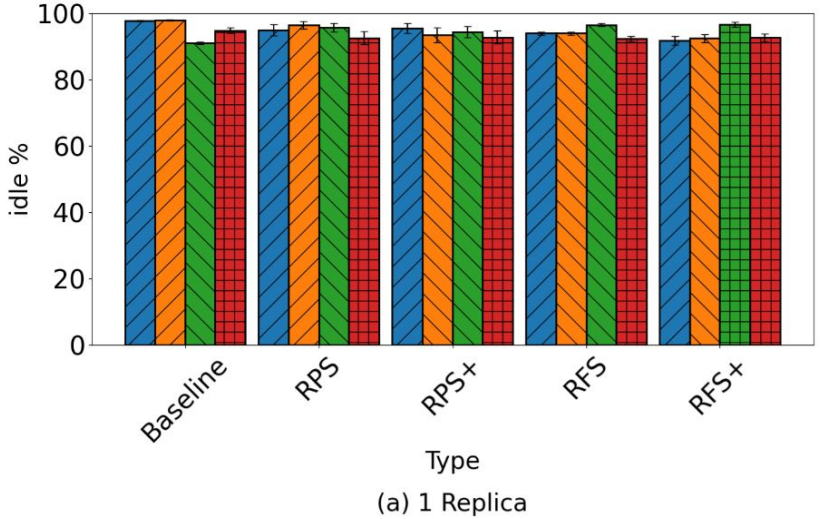
Experimental Setup



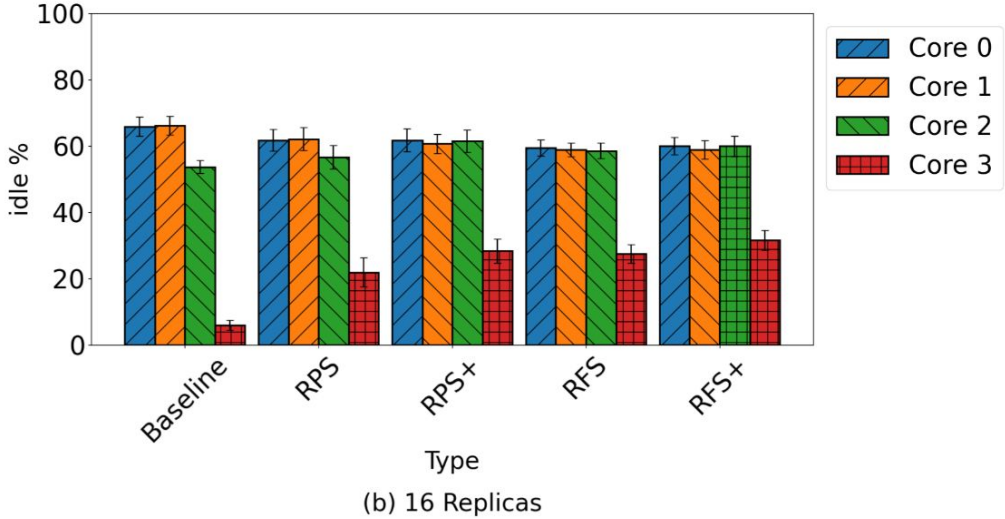
Bitrate Results



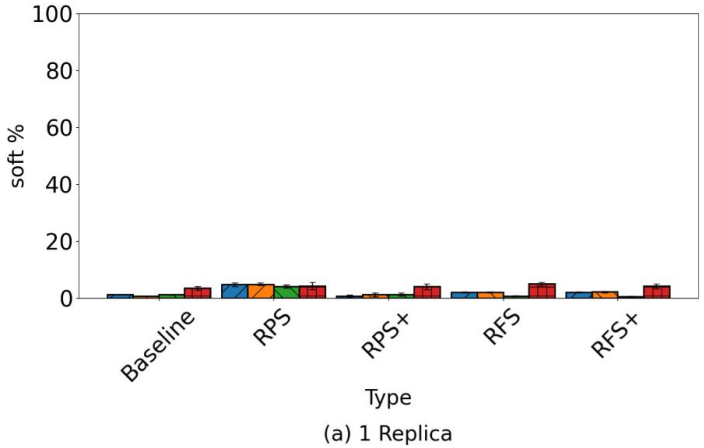
CPU Metrics



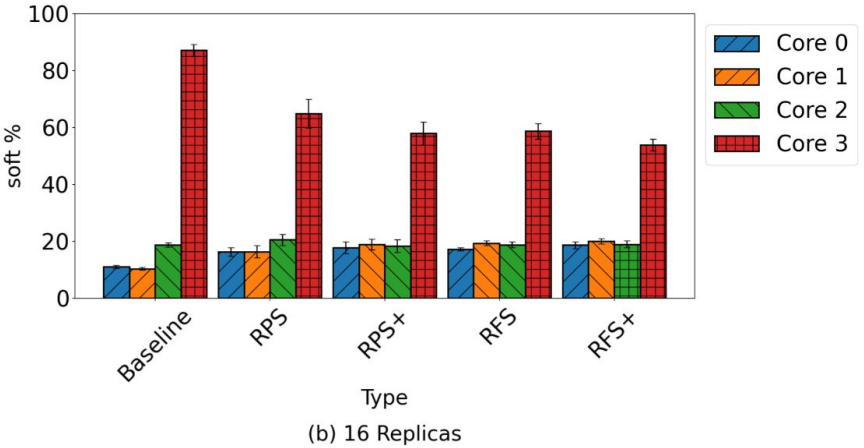
(a) 1 Replica



(b) 16 Replicas



(a) 1 Replica



(b) 16 Replicas

Conclusions

- Largest bitrate improvement seen when RFS enabled at host + container levels
  - Average increase of up to 24% over the baseline
- Enabling optimizations only at host show marginal improvements
  - ~8-9% over the baseline
- Generally, more replicas = greater improvement in bitrate with optimizations
- RFS at host + container level sees the most balanced softirq core distribution
- Overlay networks have disproportionately lower non-protocol level processing
  - RFS and RPS cannot be utilized to the full potential

Future Work

- Explore optimizations with UDP, as they function differently based on protocol
- Study efficacy using different encapsulation protocols
- Evaluate non-overlay network CNI optimizations