The Impact of Kernel Asynchronous APIs on the Performance of a Kernel VPN

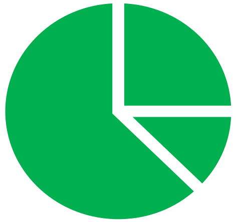
Honoré Césaire Mounah*^ Djob Mvondo*^ Julia Lawall^ David Bromberg*^
*Univ. Rennes, CNRS, IRISA
^Inria
France

9 September 2025

Context and Motivation

Virtual Private Networks (VPNS): For privacy and security

$68.3 Billion Market Share (2025)[1]



More than 1.75 Billions of users[2]

WireGuard: modern, fast VPN in the Linux Kernel

WIREGUARD
FAST, MODERN, SECURE VPN TUNNEL

Is Wireguard, a multi-threaded VPN kernel module able to handle thousands of clients efficiently?

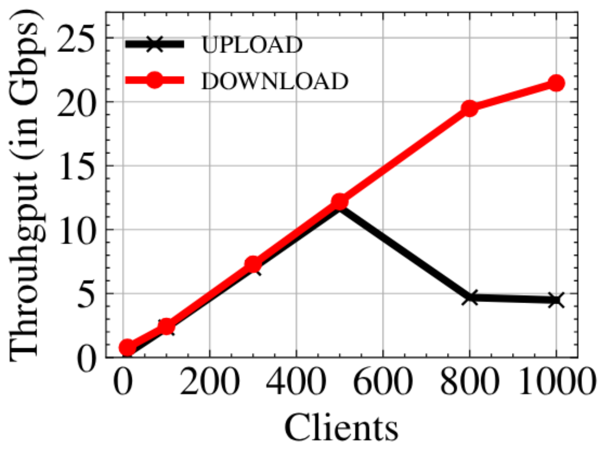
[1]https://www.coherentmarketinsights.com/industry-reports/virtual-private-network-market

[2]https://surfshark.com/blog/vpn-users

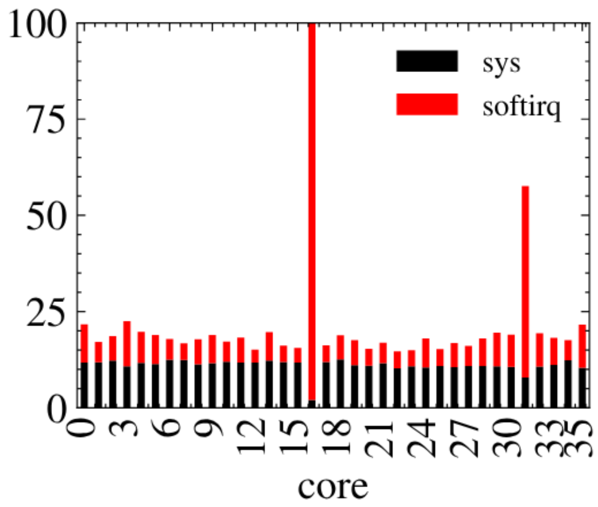
Question: Does WireGuard scale?

- Evaluation:
  - 1,000 clients * 25Mbps = 25 Gbps generated traffic
  - Use cases: Client Upload, Client Download
  - 25 Gbps Mellanox Connect-X 4 NIC
  - 18 Cores Intel Xeon Gold 5220
- Metrics
  - The forwarded network throughput by the server
  - The Server CPU Usage

Problem: WireGuard doesn't scale!



Reception Throughput



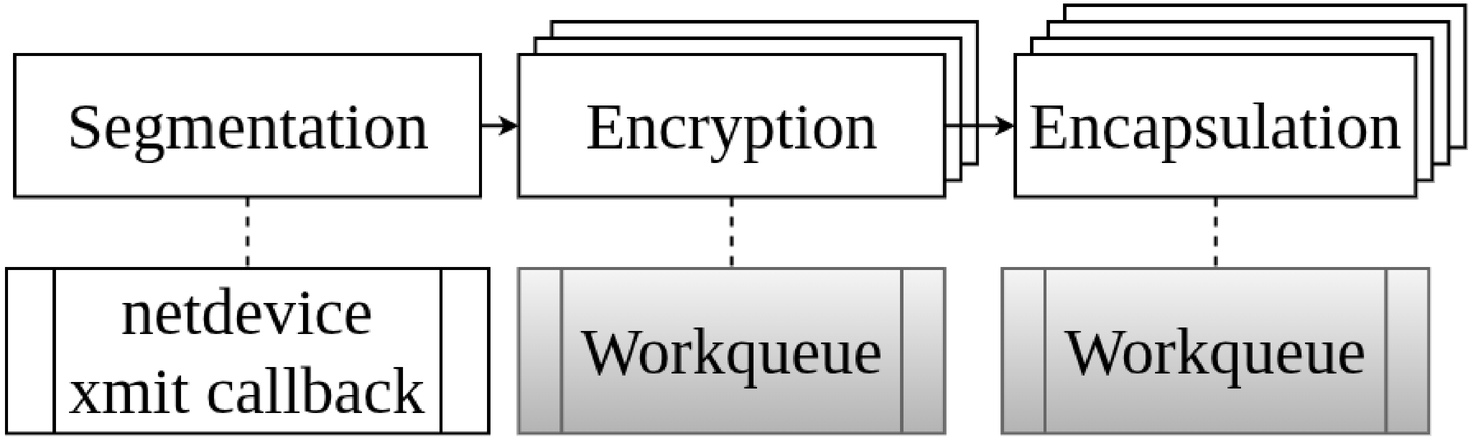
CPU usage

- Download use case scales well
- **Upload use case doesn't scale!**
  - Peaks at 12 Gbps with 500 clients
  - Plateaus at 4.5 Gbps (500+ clients)
- CPU usage is not 100% at 1,000 clients
  - *CPU is not the bottleneck*
  - *NIC is not the bottleneck*

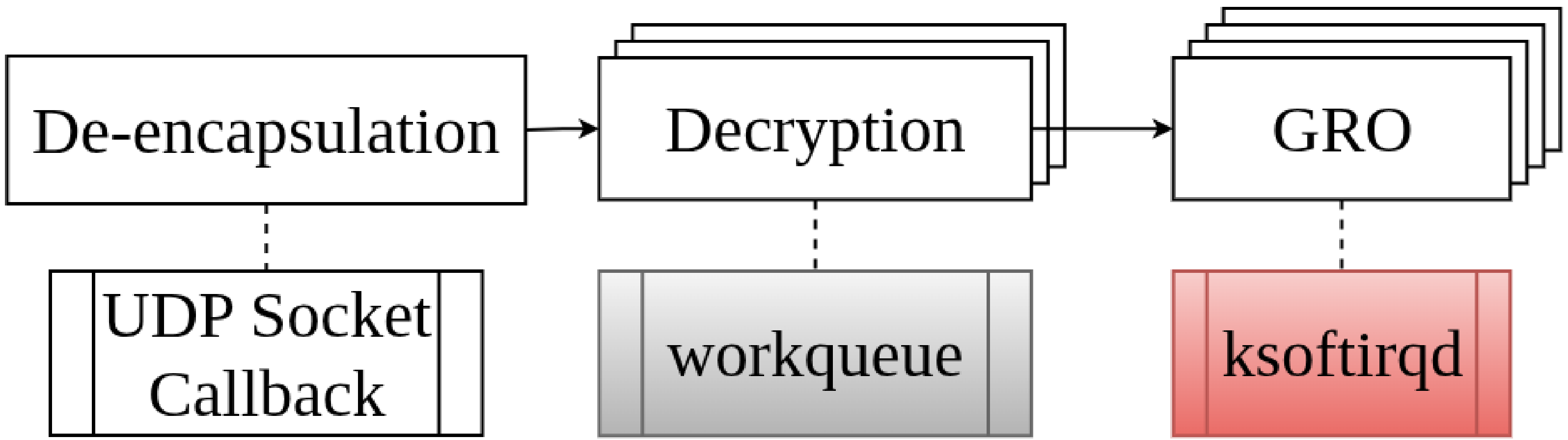
WireGuard Linux Kernel Implementation

- Performs:
  - Traffic Encryption
  - UDP Tunnelling
- Implemented on top of Linux network stack
  - Leverages Generic Segmentation/Receive Offload (GSO/GRO)
- Uses Linux kernel asynchronous APIs:
  - Workqueues
  - Softirq
- Two pipelines:
  - Transmission pipeline
  - Reception pipeline

Transmission Pipeline

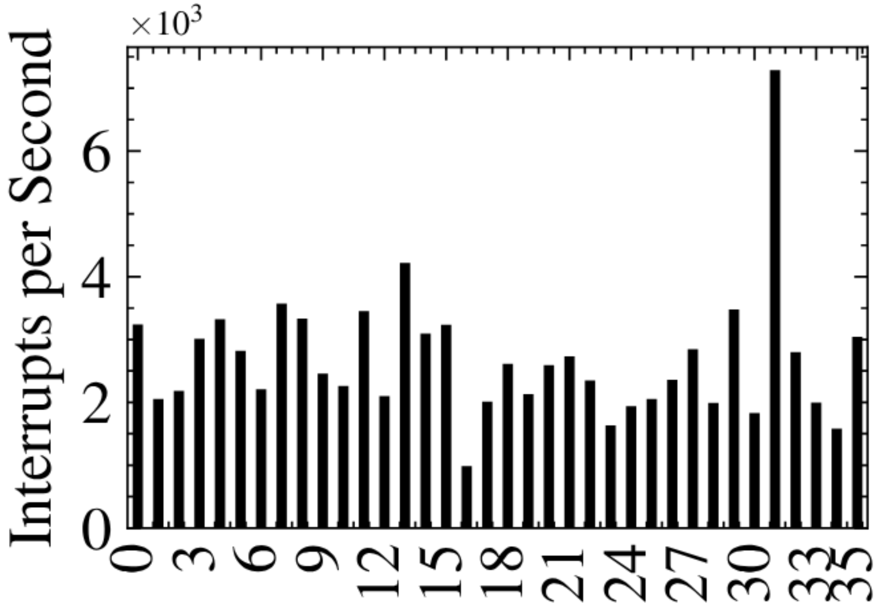


Reception Pipeline



Bottleneck is not trivial!

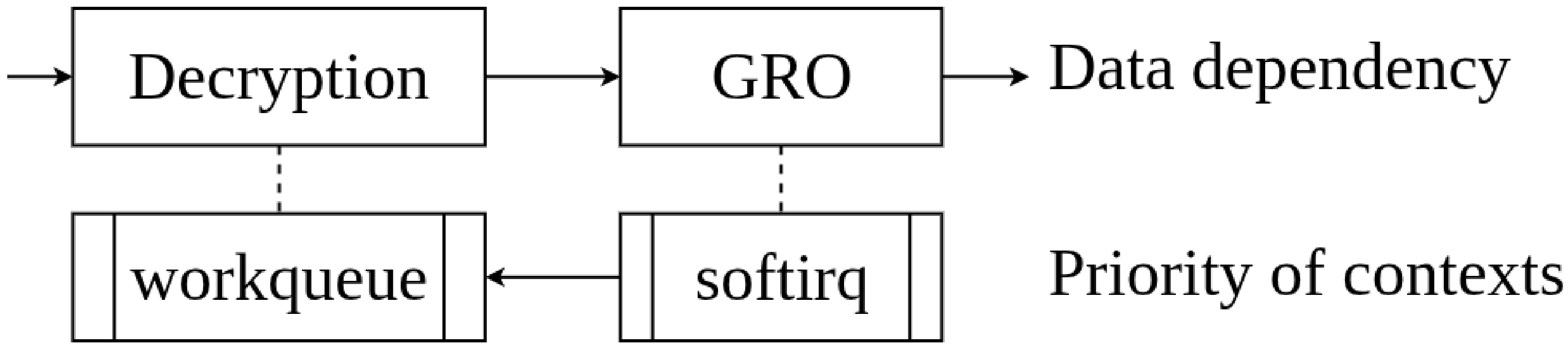
- Checked multiple possible reasons:
  - Network packet distribution across CPUs (RSS) works fine
  - Workqueues are fine, as there is no problem in the Download use case where transmission pipeline is sollicited
  - WireGuard is multi-threaded



Net RX Interrupts distribution

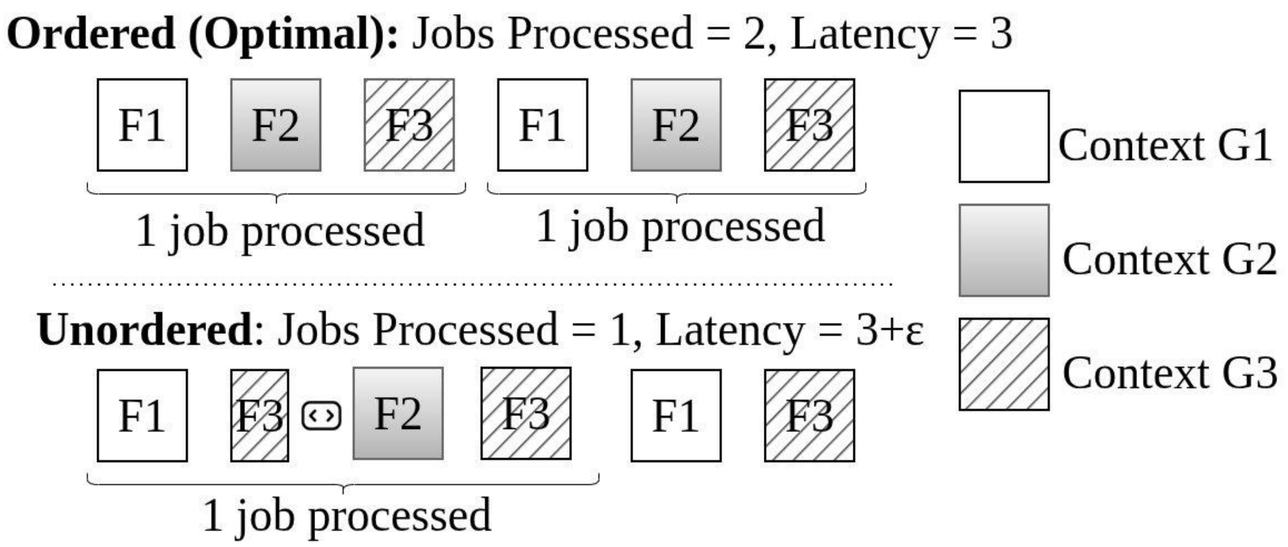
So what is the real problem?

The Bottleneck is Execution Order Inversion



Definition: later pipeline stages preempt earlier ones due to priority mismatch

Impact of EoI



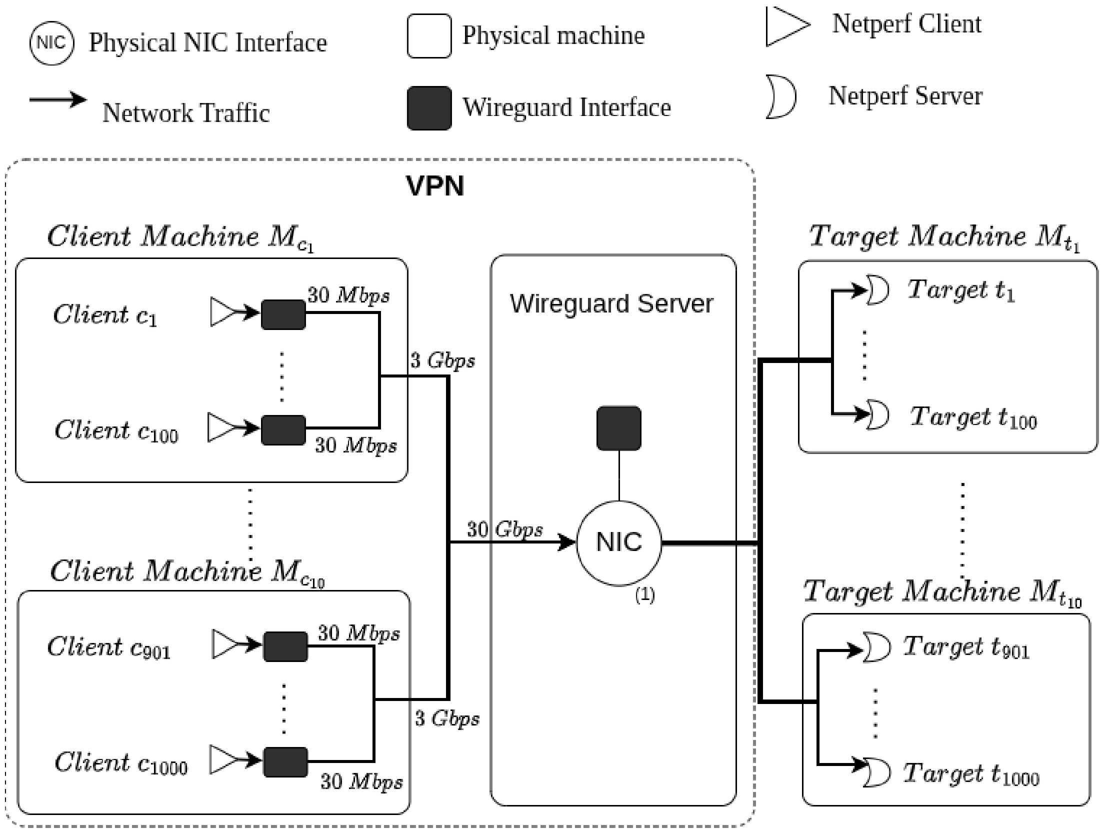
- EoI happens in 80% of all the jobs´ processing
- EoI increases latency and decreases throughput

Solution: Using Different Asynchronous APIs

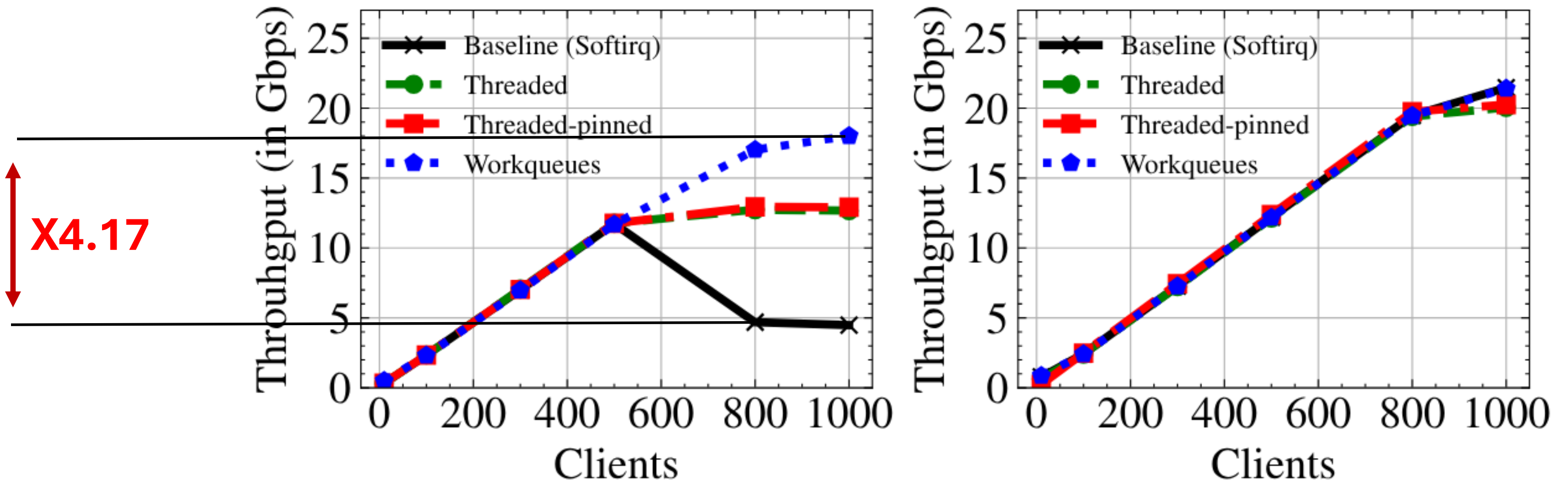
- **Run GRO at the same priority as decryption to preserve order**
- Two alternatives for GRO execution:
  - Kthreads (threaded NAPI)
  - Workqueues (new extension to NAPI)
- Kthreads
  - Easy to deploy (config only)
  - But one thread per client -> scalability issues
- Workqueues
  - Requires kernel + Wireguard changes
  - Fixed thread pool (per CPU) -> Scalable

Evaluation and Results

- Evaluation Setup:
  - Testbed: 21 servers
  - Intel Xeon Gold 5220 (18 cores)
  - 25 Gbps NIC, Mellanox Connect-X 4
  - Linux 6.1 (LTS), Debian 12
- Evaluation Scenario
  - Up to 1,000 clients generating each 25Mbps of traffic upload and download with iPerf3
- Metrics:
  - Throughput, CPU Usage, 99th Tail Latency



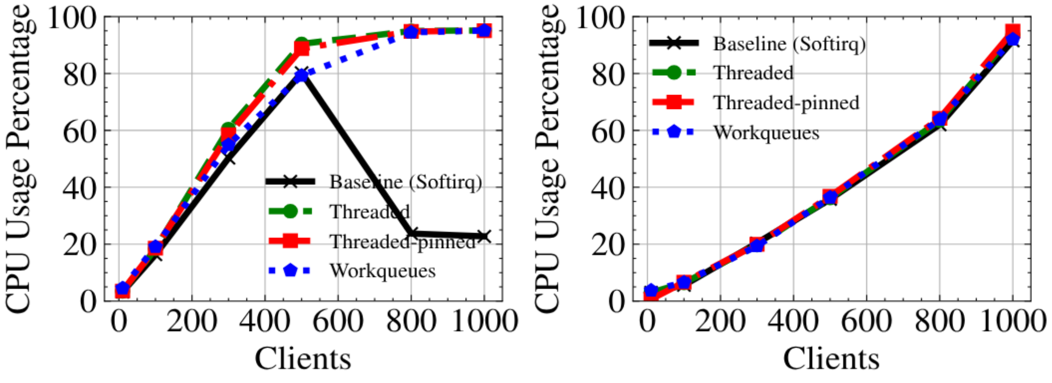
Evaluation and Results: Throughput



(a) Reception **(b) Transmission**

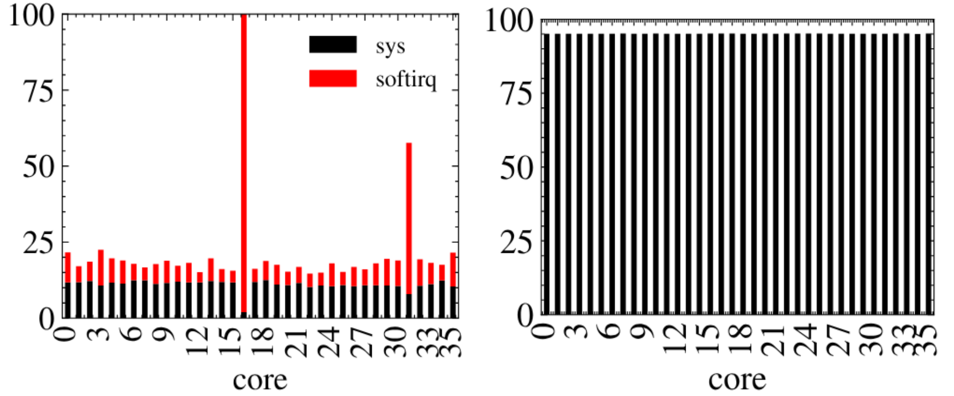
- In reception, with 800+ clients, throuhgput:
  - remains at 12.5 Gbps with kthreads (a x2.8 Improvement)
  - Scales with workqueueus up to 18.8 Gbps
    - **A x4.17 improvement**
- Transmission pipeline is not impacted, which is good.

Evaluation and Results: CPU Usage



(a) Reception **(b) Transmission**

CPU Usage



(a) Baseline (softirq) **(b) kthreads/workqueues**

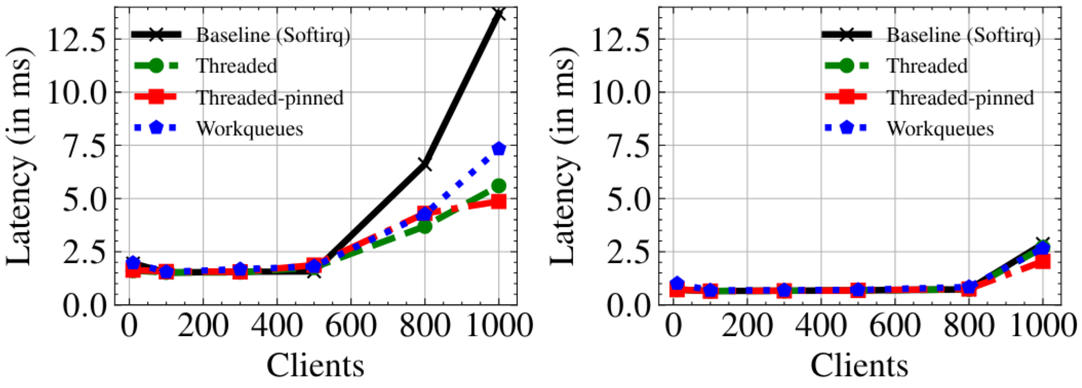
Per core CPU usage

- CPU is now fully used

Evaluation and Results: Latency

- **Upload:**
  - Baseline 13.7 ms
  - kthreads 5.6 ms (4.8 ms pinned, best)
  - workqueue 7.3 ms.
- **Download:**
  - Low latency overall (0.6–0.7 ms up to 500 clients)
  - rises at scale (baseline 2.8 ms, kthreads/workqueue 2.7 ms, pinned 2.0 ms, best)

Pinned kthreads consistently achieve the lowest latency, especially under high load.



(a) Client upload (b) Client download

99th Tail Latency

Takeaways

When designing a multi-threaded asynchronous application, the choice of which execution context to use is crucial, even more so for kernel modules.