Host Networking (Google Case Study)

ECE/CS598HPN

Radhika Mittal

Snap: a Microkernel Approach to Host Networking

SOSP'19

Slides largely borrowed from the SOSP talk

Summary

Snap: Framework for developing and deploying packet processing software

- Goals: Performance and Deployment Velocity
- Technique: Microkernel-inspired userspace approach

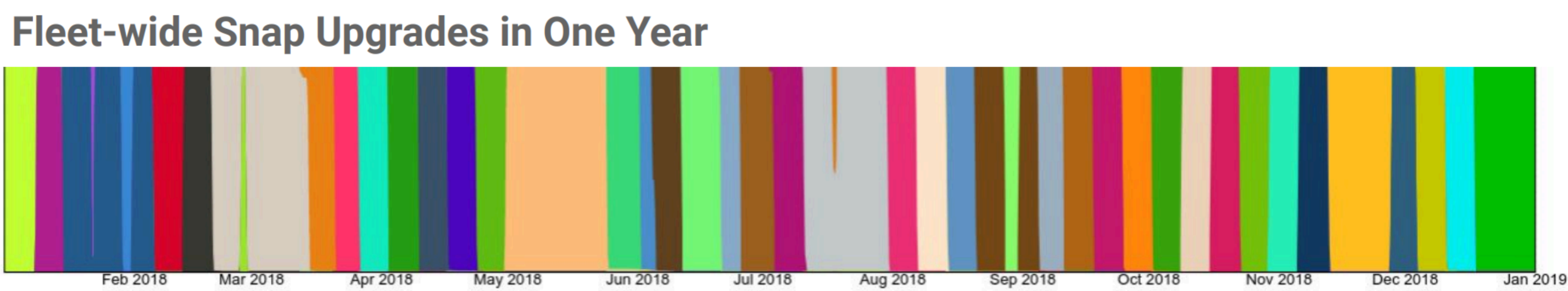
Snap supports multiple use cases:

- Andromeda: Network virtualization for Google Cloud Platform [NSDI 2018]
- Espresso: Edge networking [SIGCOMM 2017]
- Traffic shaping for Bandwidth Enforcement
- New: High-performance host communication with "Pony Express"

3x throughput efficiency (vs kernel TCP), 5M IOPS, and weekly releases

Motivation

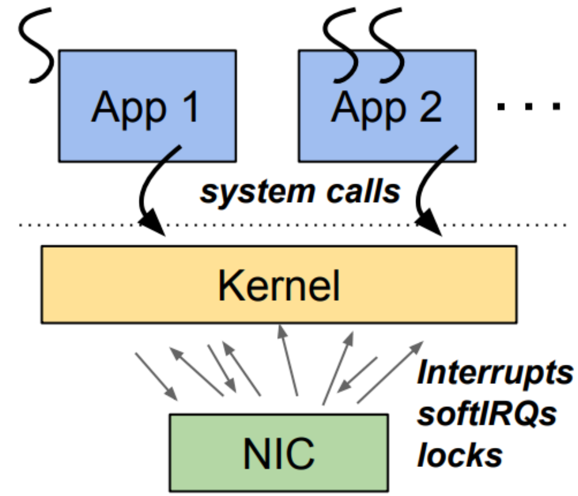
- Growing performance-demanding packet processing needs at Google
- The ability to rapidly **develop and deploy** new features is just as important!



Monolithic (Linux) Kernel

Deployment Velocity:

- Smaller pool of software developers
- More challenging development environment
- Must drain and reboot a machine to roll out new version
- Typically months to release new feature



Performance:

- Overheads from system calls, fine-grained synchronization, interrupts, and more.

LibraryOS and OS Bypass

Networking logic in application binaries

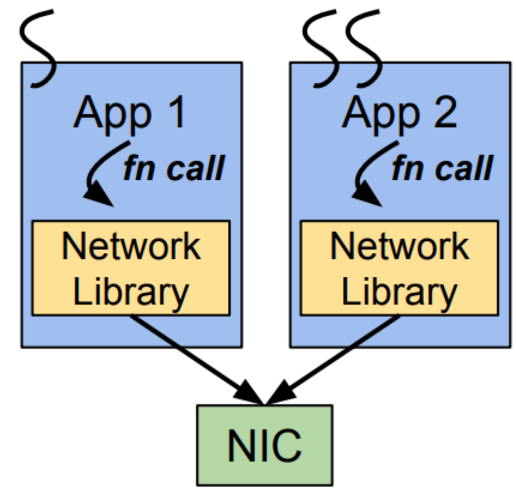
Examples: Arrakis, mTCP, Ix, ZygOS, and more

Deployment Velocity:

- Difficult to release changes to the fleet
- App binaries may go months between releases

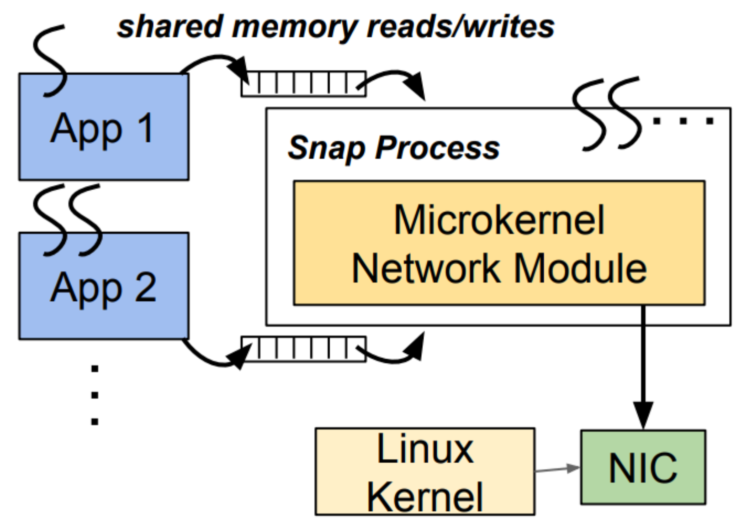
Performance:

- Can be very fast
- But typically requires spin-polling in every application
- Benefits of centralization (i.e., scheduling) lost
  - Delegates all policy to NIC



Microkernel Approach

Hoists functionality to a separate userspace process



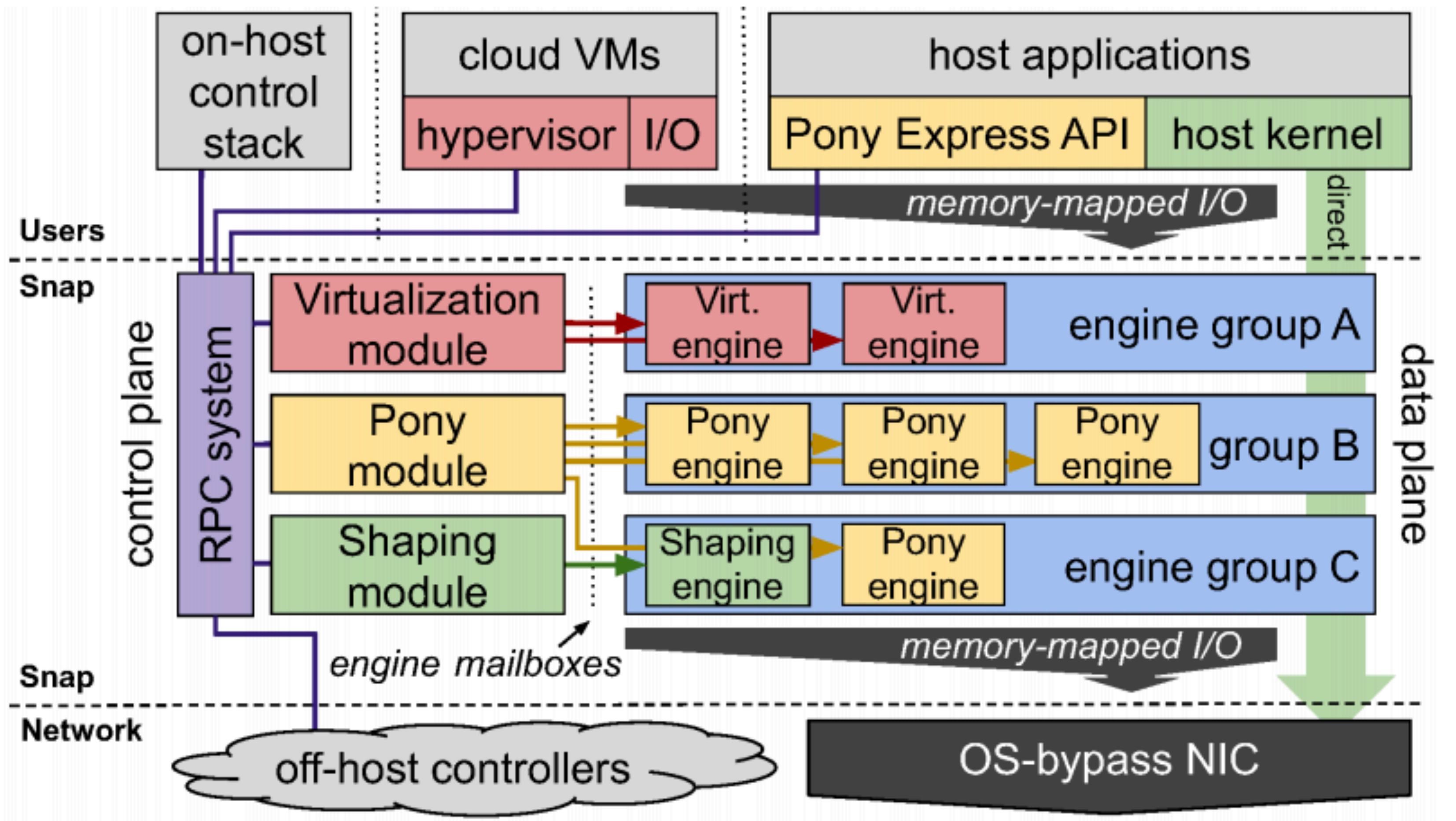
Deployment Velocity:

- Decouples release cycles from application and kernel binaries
- Transparent upgrade with iterative state transfer

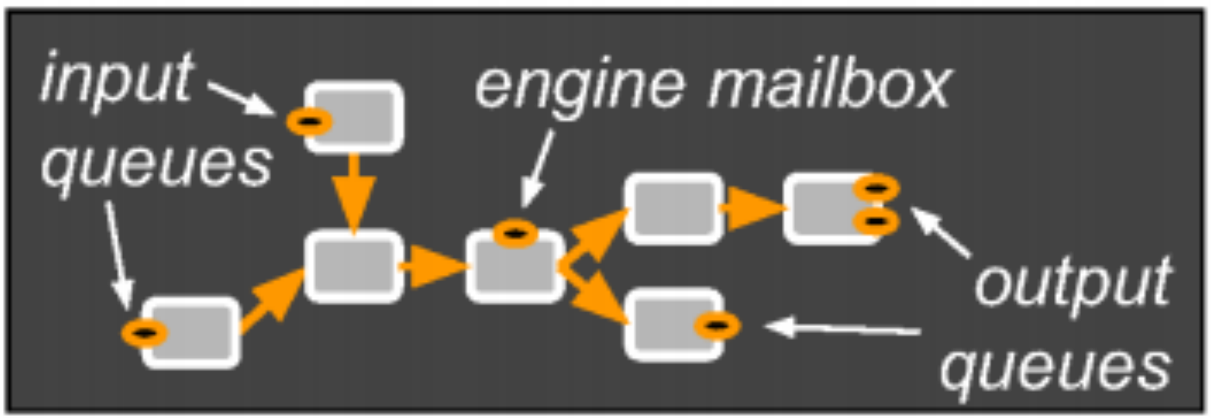
Performance:

- Fast! Leverages kernel bypass and many-core CPUs
- Maintains centralization of a kernel
- Can implement rich scheduling/multiplexing policies

Snap Architecture



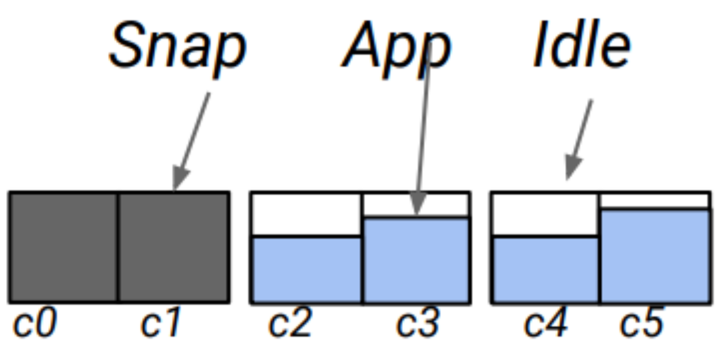
Snap Engine



Snap Engine Scheduling Modes

Dedicated Cores

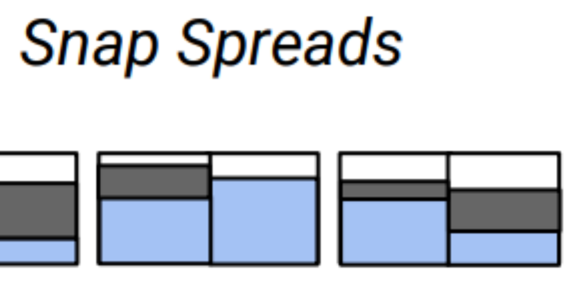
- Static provisioning of N cores to run engines
- Simple and best for some situations.
- Provisioning for the worst-case is wasteful
- Provisioning for the average case leads to high tail latency



Snap Engine Scheduling Modes

Spreading Engines

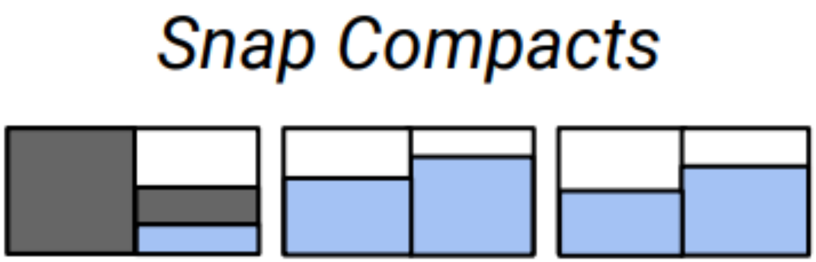
- Bind each engine to a unique kernel thread
- Interrupts triggered from NIC or application to schedule on-demand
- Leverages new micro-quanta kernel scheduling class for tighter latency
- *Can* provide best tail latency
- Scheduling pathologies and overheads



Snap Engine Scheduling Modes

Compacting Engines

– Compacts engines to as few cores as possible
– Periodic polling of queuing delays to re-balance engines to more cores
– *Can* provide best CPU efficiency.
– Timely detection queue build-up.



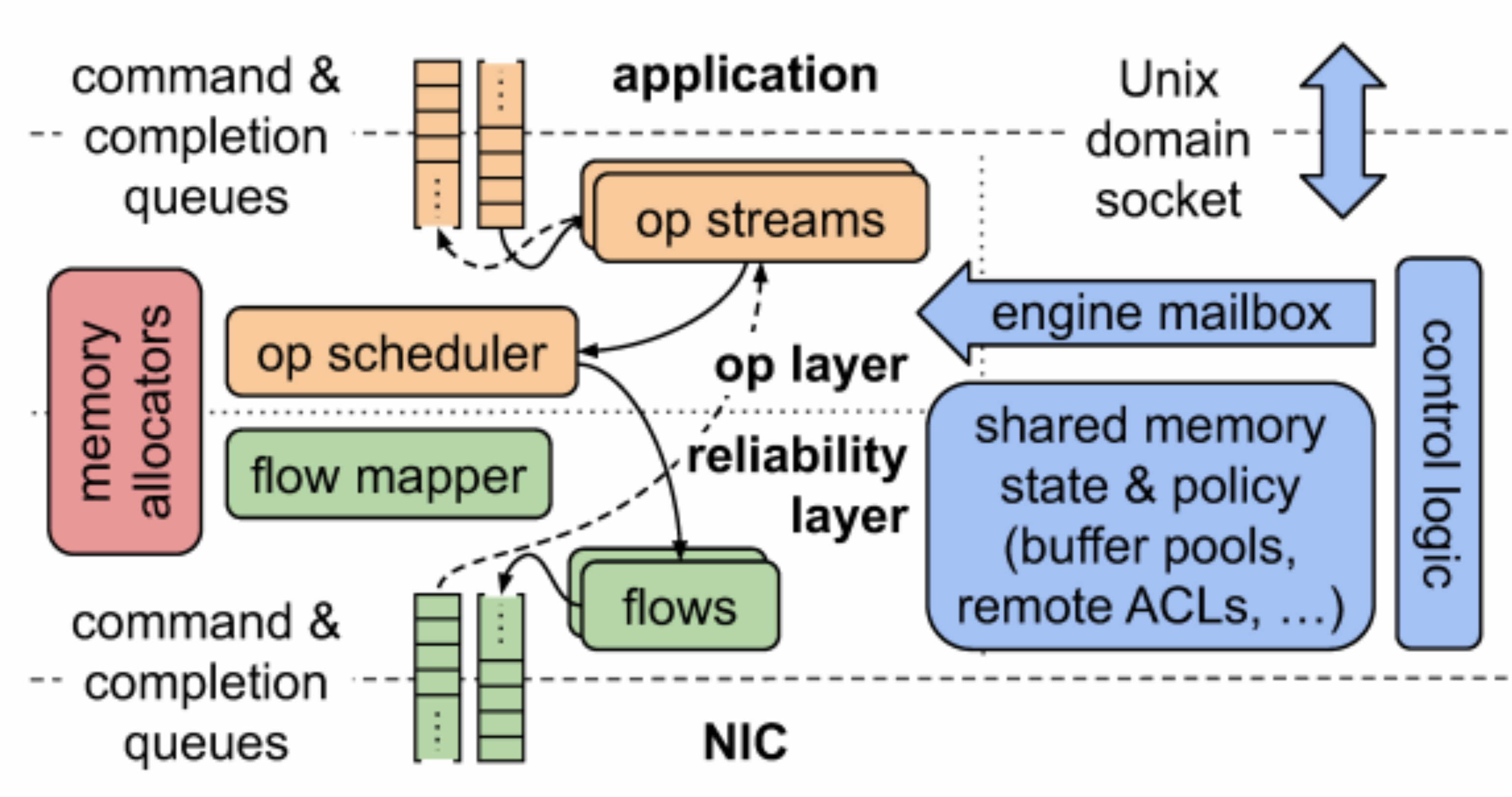
High Performance Communication

Pony Express Communication Stack

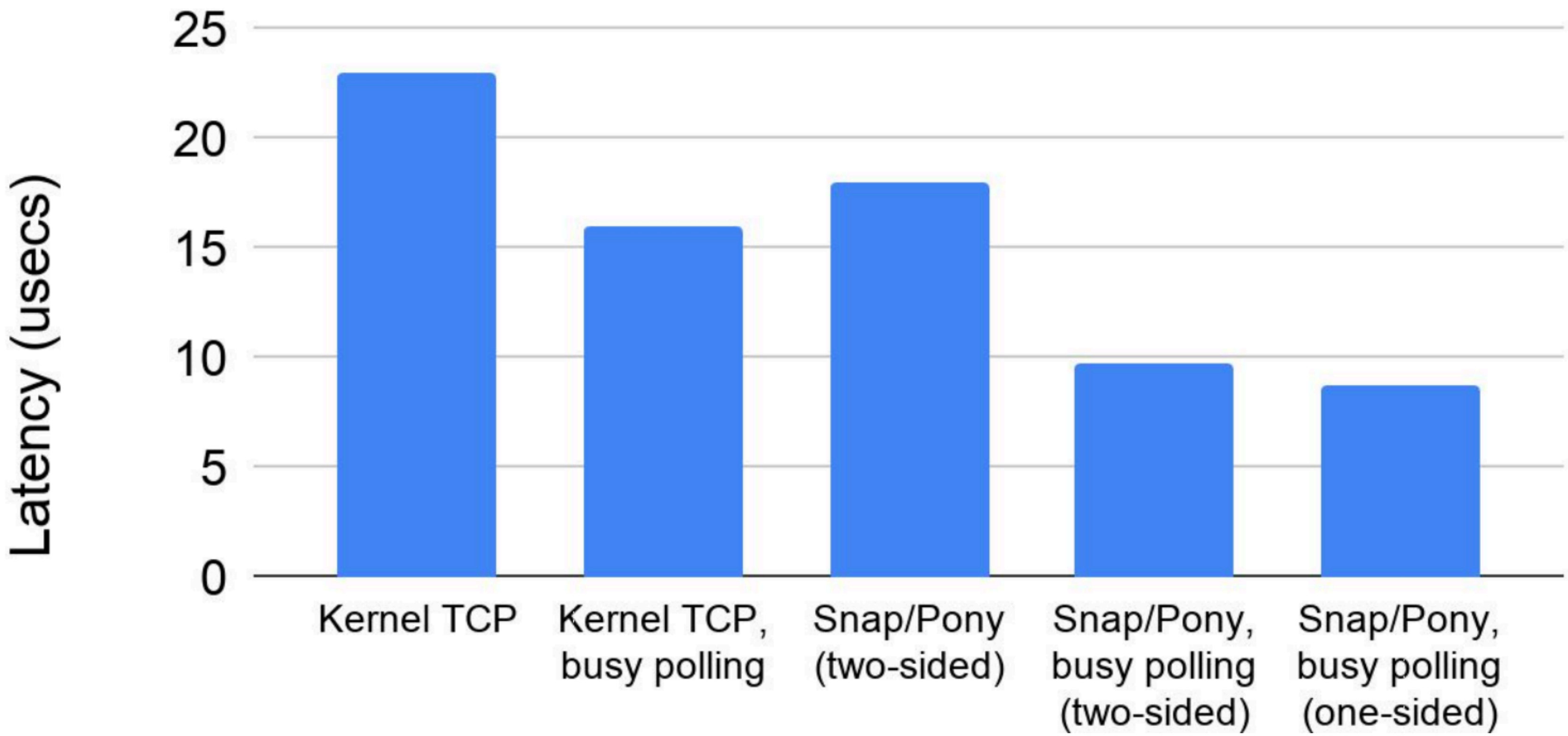
- Implement a full-fledged reliable transport and interface
  - RDMA-like operation interface to applications
  - Two-sided for classic RPC
  - One-sided (pseudo RDMA) operations for avoiding invocation of application thread scheduler
  - Custom one-sided operations to avoid shortcomings of RDMA (i.e., pointer chase over fabric)
  - Custom transport and delay-based congestion control (Timely)

High Performance Communication

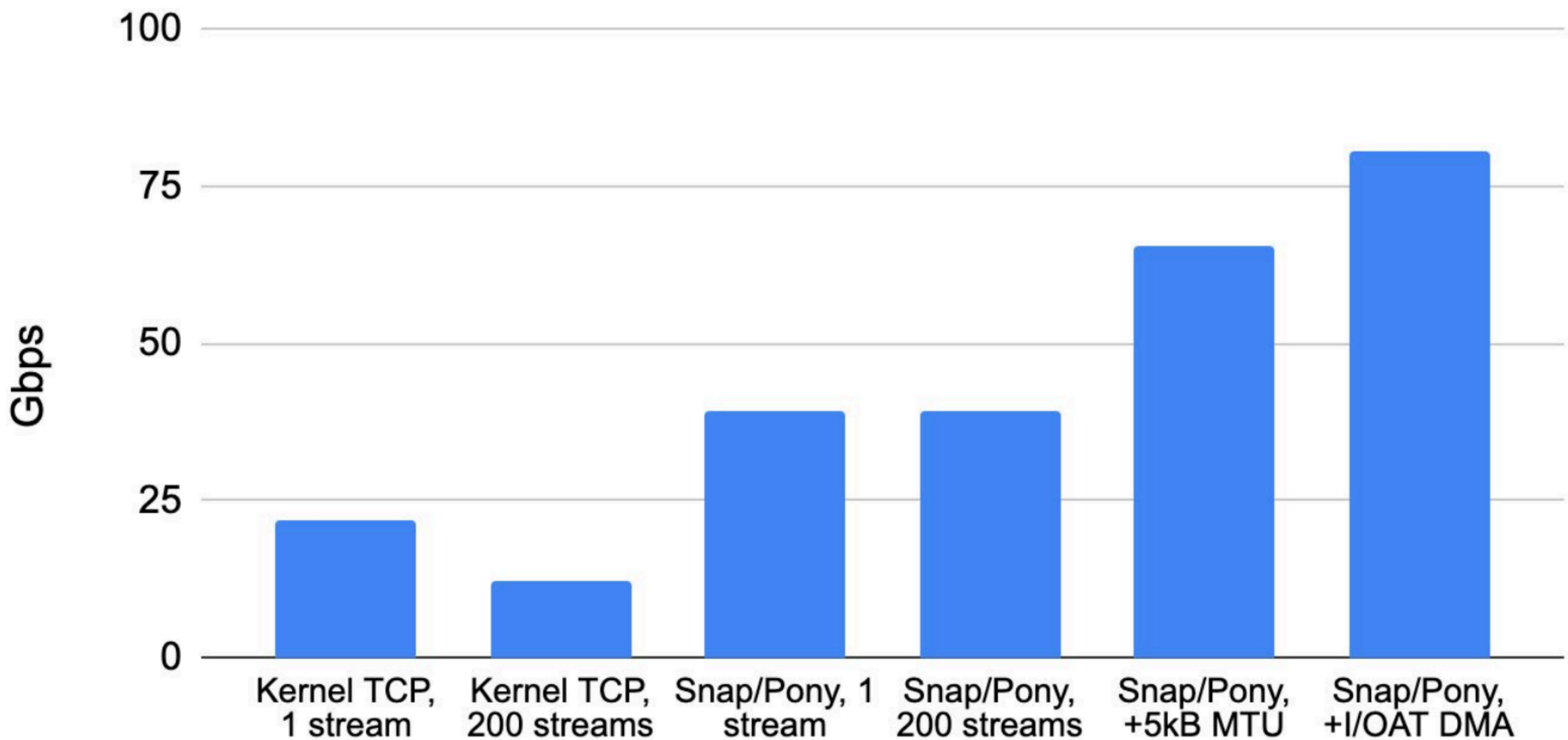
Pony Express Communication Stack



Evaluation: Ping-pong latency



Evaluation: Throughput



Evaluation: Comparison with RDMA

- Switching to Pony Express "doubled the production performance of the data analytics service".
- Stringent RDMA rate limits applied to prevent NIC cache overflow, and ensuing PFCs.
- Could be disabled with Pony Express.

Your Opinions

Pros:

- Diverse services (virtualization, packet processing, shaping)
- More sophisticated CPU scheduling (compared to earlier works)
- Deployed (and tested) in production clusters over many years.
- Focus on transparent upgrades and fast development cycles.

Your Opinions

Cons:

- Performance trade-offs over LibraryOS based appproaches.
- How to use SNAP in multi-tenant settings?
- How to handle failure or rollback during upgrades?
- API incompatibility
- Designing and configuring engines could be tricky.
- Security story seems a bit unconvincing
- Unconvincing flow control for one-sided operations.
- Context-switching overhead between PonyExpress and application.

Your Opinions

Ideas:

- Can PonyExpress be extended to transport outside of datacenters?
- Synchronous API over Snap?
- Better scheduling and scaling for CPU
- Is Snap is a good for IoT/edge devices?
- Support multi-threaded Snap engines
- Comparison with other transport stacks.