Snap: a Microkernel Approach to Host Networking

Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, *Carlo Contavalli*, Michael Dalton*, Nandita Dukkipati,* William C. Evans, *Steve Gribble,* Nicholas Kidd, *Roman Kononov, Gautam Kumar,* Carl Mauer, Emily Musick, Lena Olson, *Erik Rubow,* Michael Ryan, Kevin Springborn, *Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat*

Google-Madison and *Google-Sunnyvale*

** work performed while at Google*

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

Outline

Motivation

Design

Evaluation

Experiences and Challenges

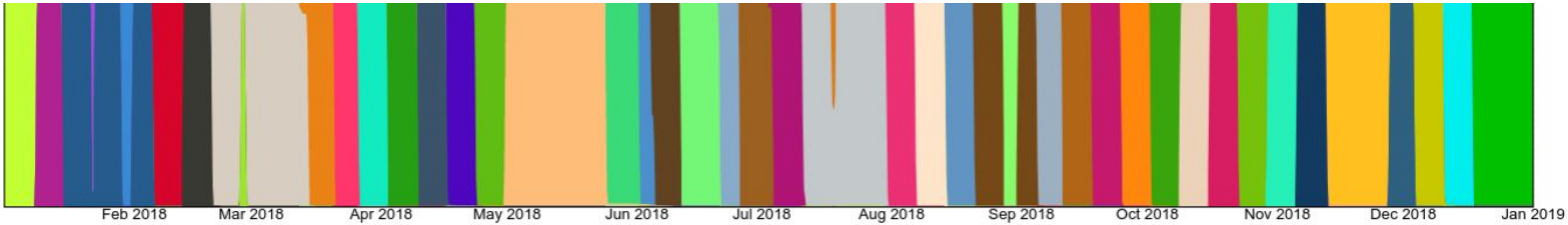
Conclusion

Motivation

Growing performance-demanding packet processing needs at Google

The ability to rapidly **develop and deploy** new features is just as important!

Fleet-wide Snap Upgrades in One Year



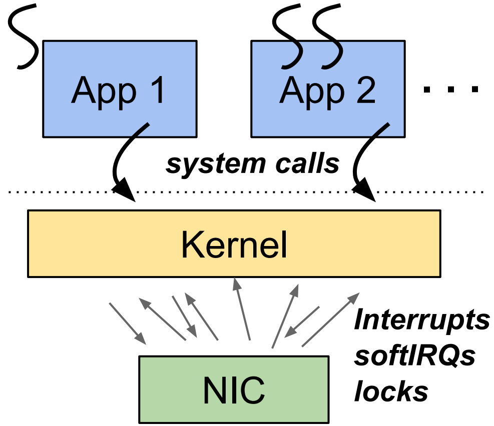
Monolithic (Linux) Kernel

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

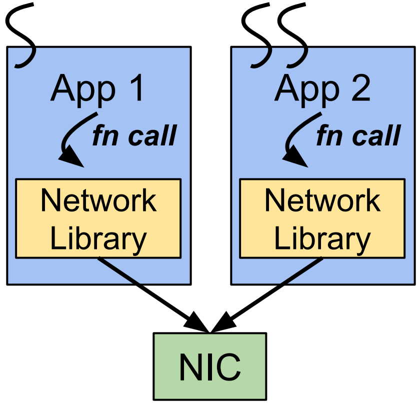
Networking logic in application binaries

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



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

Microkernel Approach

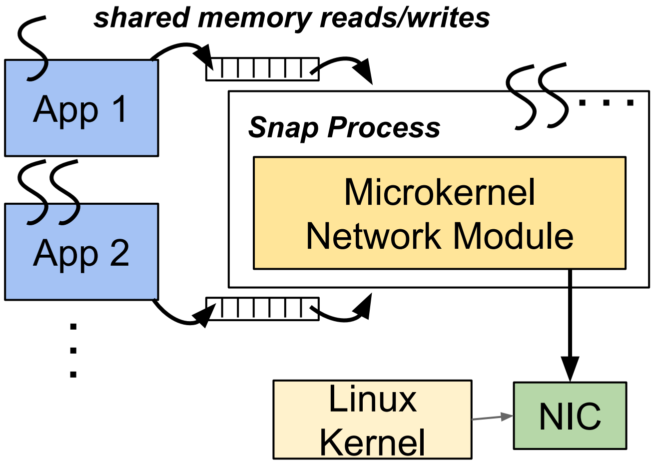
Hoists functionality to a separate userspace process

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



Outline

Motivation

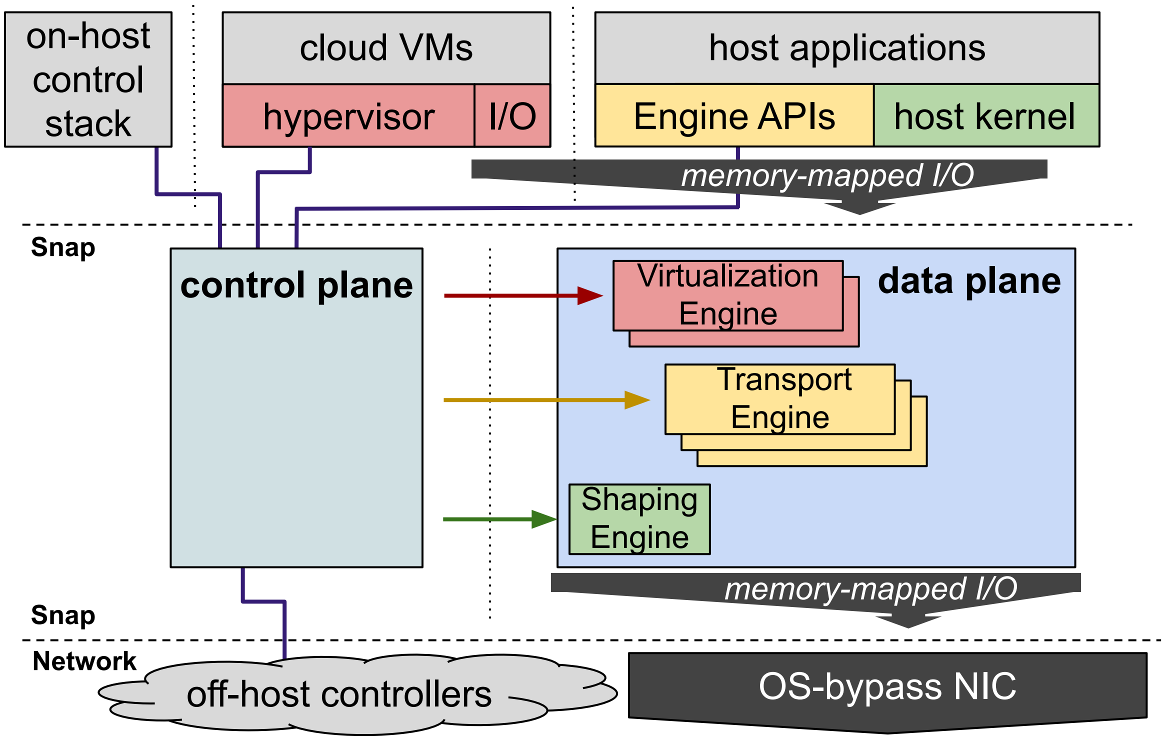
Design

Evaluation

Experience and Challenges

Conclusion

Snap Architecture Overview



Snap Engine

- Key dataplane element
- Implements packet processing pipelines
- Unit of CPU scaling

Snap Engines implement a *Run()* method invoked by Engine Threads

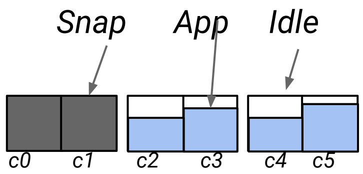
Principled Synchronization

- No blocking locks

Snap Engine Scheduling Modes

Dedicated Cores

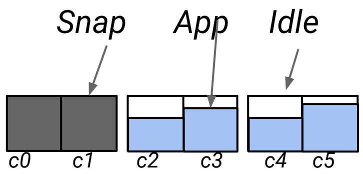
- Static provisioning of N cores to run engines
- Simple and best for some situations



Snap Engine Scheduling Modes

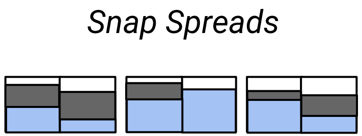
Dedicated Cores

- 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**



⇒ Need dynamic provisioning of CPU resources

Snap Engine Scheduling Modes



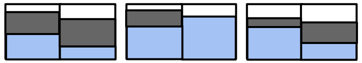
Spreading Engines

- 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

Pros: Can provide the best tail latency   Cons: scheduling pathologies and overhead

Snap Engine Scheduling Modes

Snap Spreads



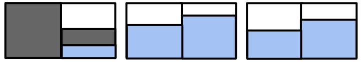
Spreading Engines

- 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

Pros: Can provide the best tail latency   Cons: scheduling pathologies and overhead

Compacting Engines

Snap Compacts



- Compacts engines to as few cores as possible
- Periodic polling of queuing delays to re-balance engines to more cores

Pros: Can provide the best CPU efficiency   Cons: detecting queue build-up when many engines

High Performance Communication with Snap

Snap enabled us to build the “Pony Express” communication stack

- Goal: high performance at Google scale

Pony Express engines 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)

Integrates into existing stacks (i.e., gRPC) and applications

Path towards seamless access of hardware offloads

Outline

Motivation

Design

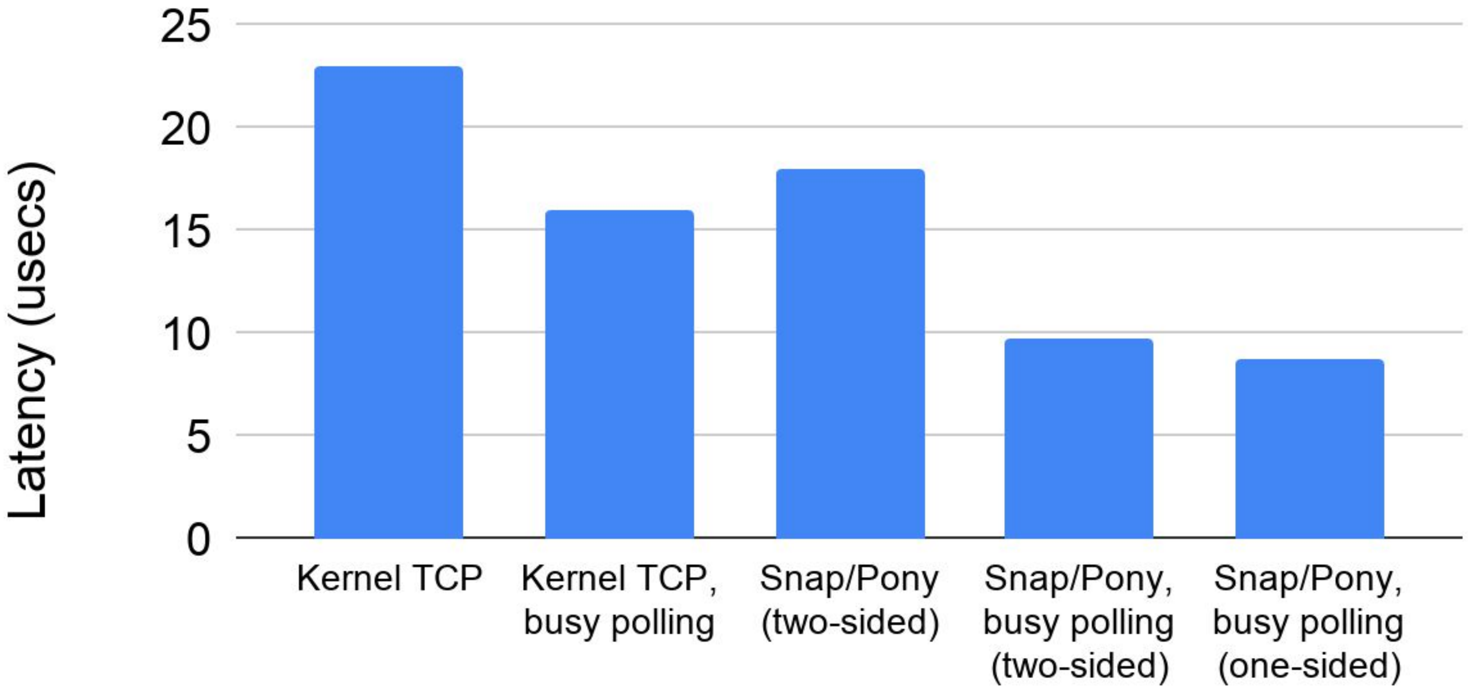
Evaluation

Experience and Challenges

Conclusion

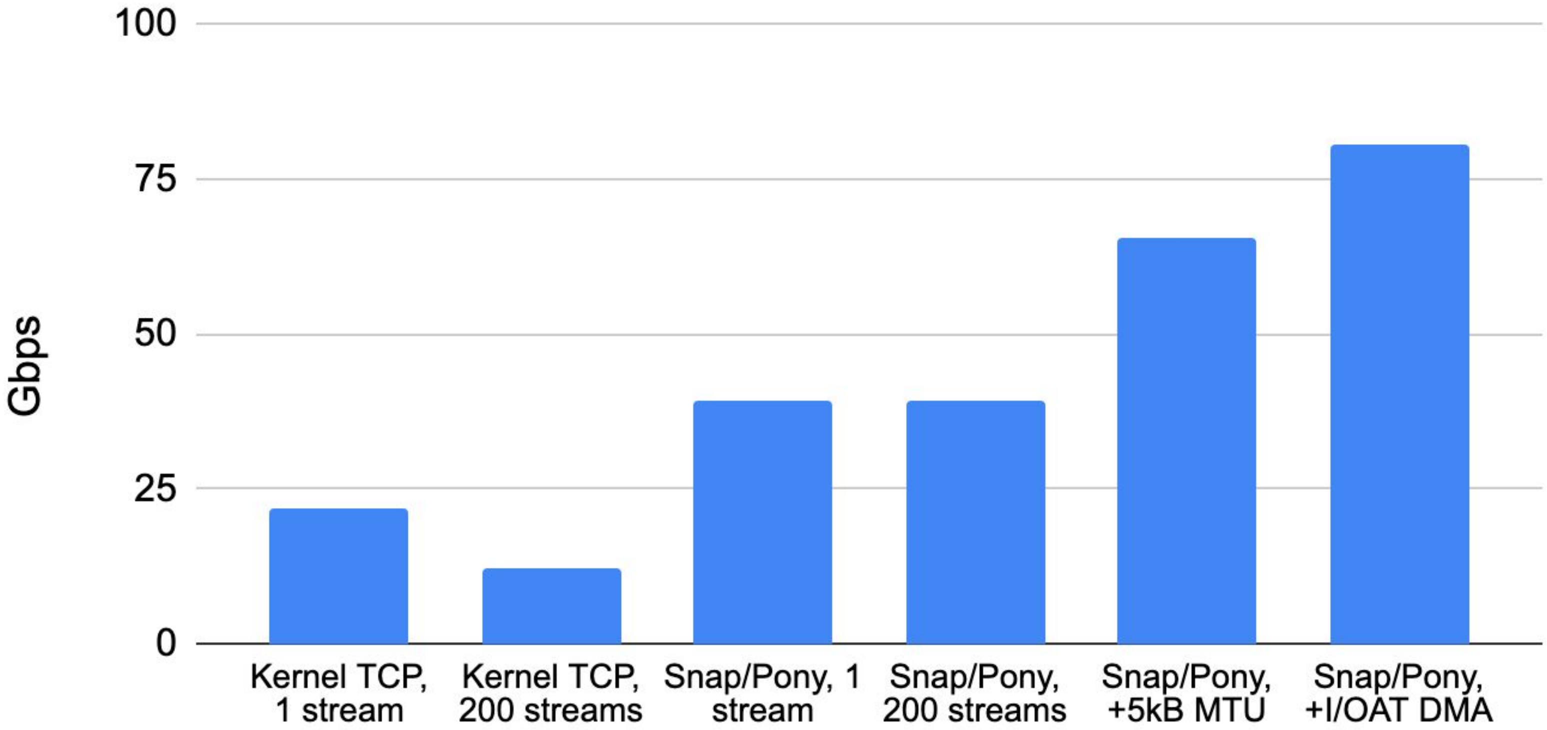
Evaluation -- Ping Pong Latency

2-node “TCP_RR”-style ping pong latency



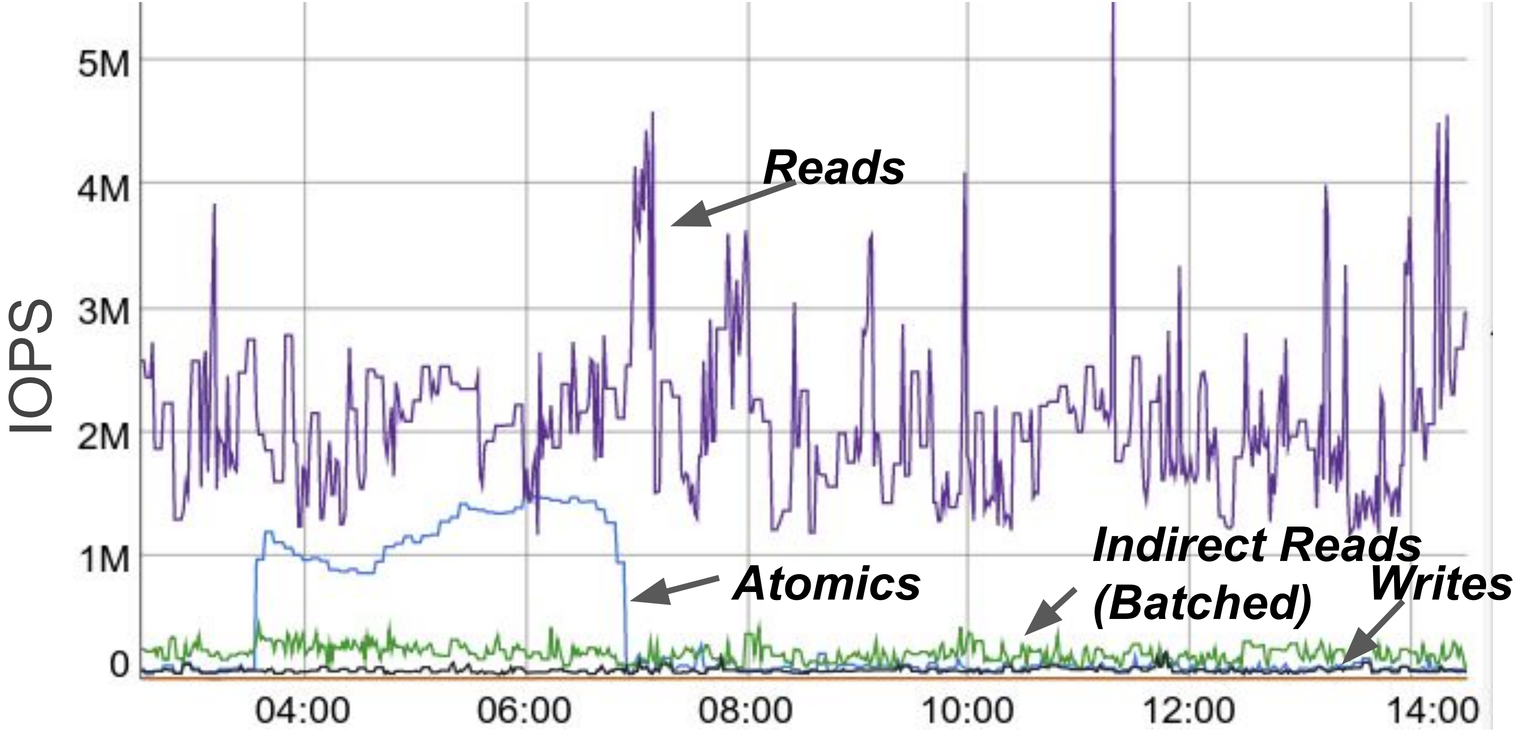
Evaluation -- Throughput

2-node "TCP_STREAM"-style throughput. Single Pony Engine, Dedicated Core



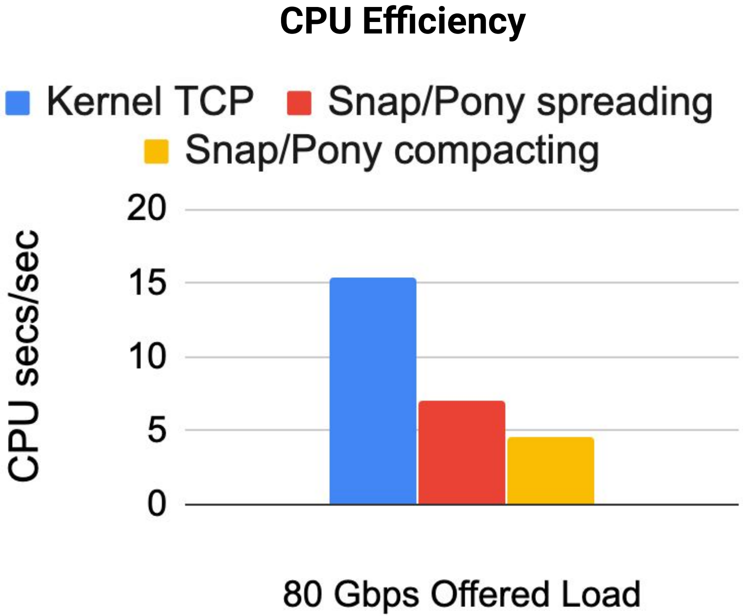
Production Dashboard of One-sided IOPS

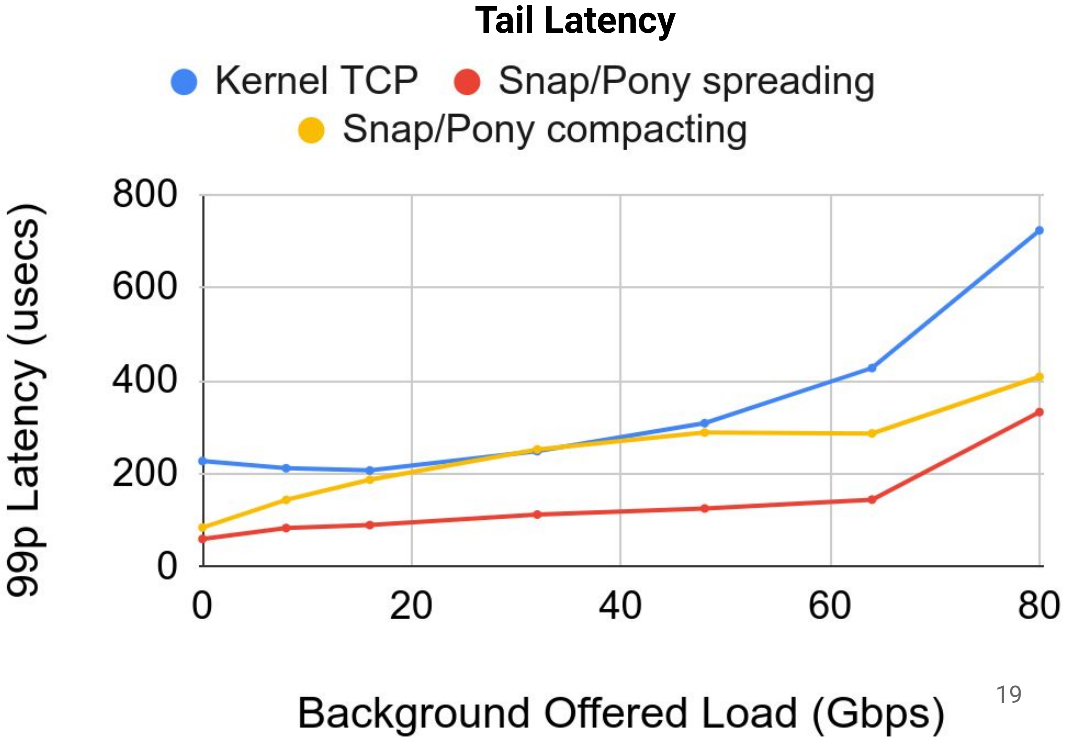
Hottest machine in one-minute intervals. Single Pony Express engine and core



Challenges with Dynamic Scaling

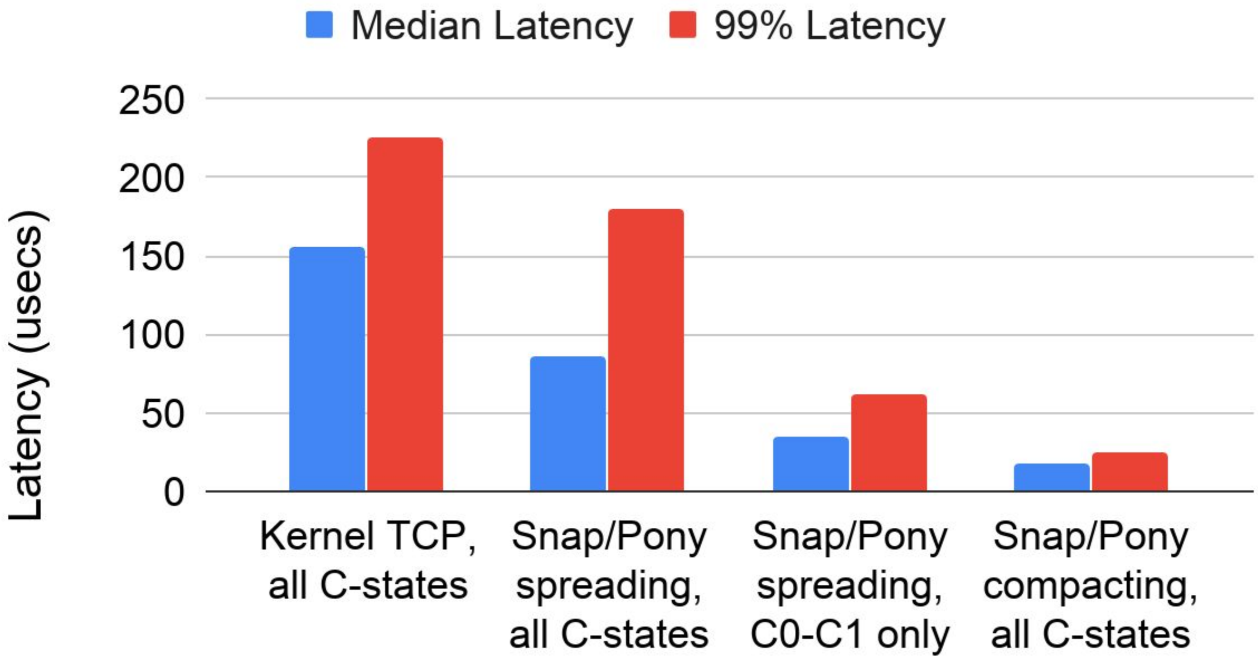
10 Pony Express Engines dynamically scheduled.





Challenges with Dynamic Scaling

Spreading engines impacted by C-states and non-preemptible kernel activity



Conclusion

Snap: a Microkernel Approach to Host Networking

- Achieves the iteration-speed advantages of userspace dev and microservices
- With the performance gains of OS bypass
- With the centralization advantages of a traditional OS kernel
- And interoperates with application threading systems and the rest of Linux