November 1, 2023 @ SoCC'23

Yama: Providing Performance Isolation for Black-Box Offloads

Tao Ji, Divyanshu Saxena, Brent E. Stephens and Aditya Akella


TEXAS
The University of Texas at Austin

THE UNIVERSITY OF UTAH

Offloading in today's datacenters

Common functions are offloaded from CPU to dedicated accelerators for better performance, power efficiency, etc.

Functions of multiple layers

- Network (L3): NAT, firewall, IDS/IPS
- Transport (L4): RDMA, TCP offload engine
- L4+ functions: Cryptography, compression
- Application (L7): Key-value cache, RPC load balancing

Most often seen on NICs – closely related to network IO

Entity-level sharing of NIC offloads

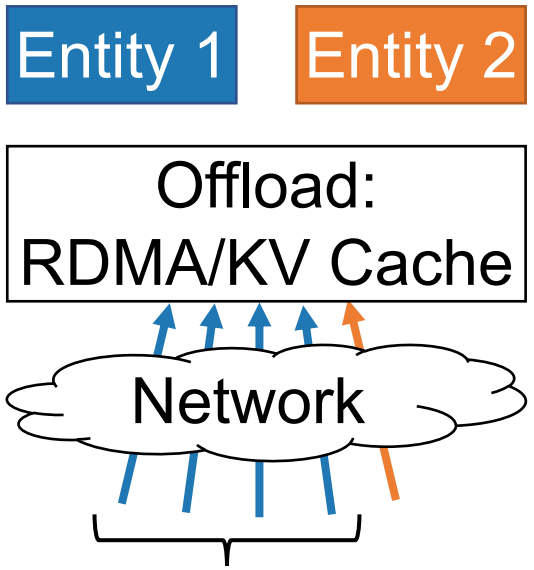
Offloads often need to be shared by high-level entities,

- e.g., first-party user/application, tenants in public clouds.
- NICs have limited capacity – hard to provision dedicated offloads to each entity.

Performance isolation with different service levels is desired.

- Similar guarantees are provided for other resources
  - e.g., link bandwidth.
- Offloads can be bottlenecks due to limited hardware performance and/or complex logic.

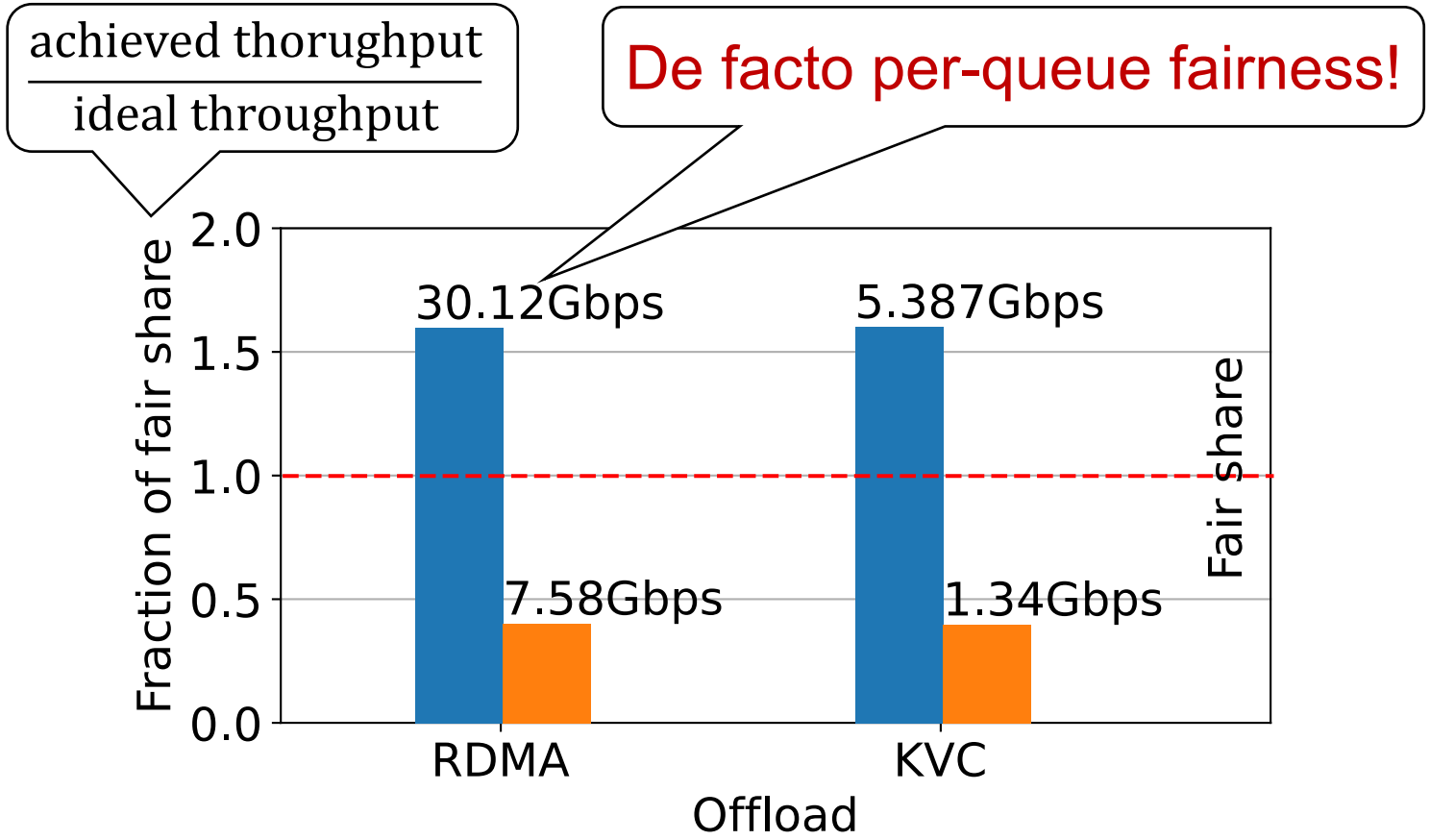
Entity-level isolation is not supported today



4x IO queues for Entity 1

Same # of CPU cores

Policy: entities share bottleneck performance equally



Problem: How to provide entity-level isolation guarantees with different service levels for existing offloads?

Target policy: weighted max-min fairness of throughput

Challenge 1: no local scheduling

Existing offload functions (and NICs) can be **black boxes**

- fixed in hardware (ASIC)
- sourced from third-party vendors
- e.g., RDMA, cryptography, compression, etc.

Such offloads are hard to update with extra mechanisms to enforce entity-level isolation locally.

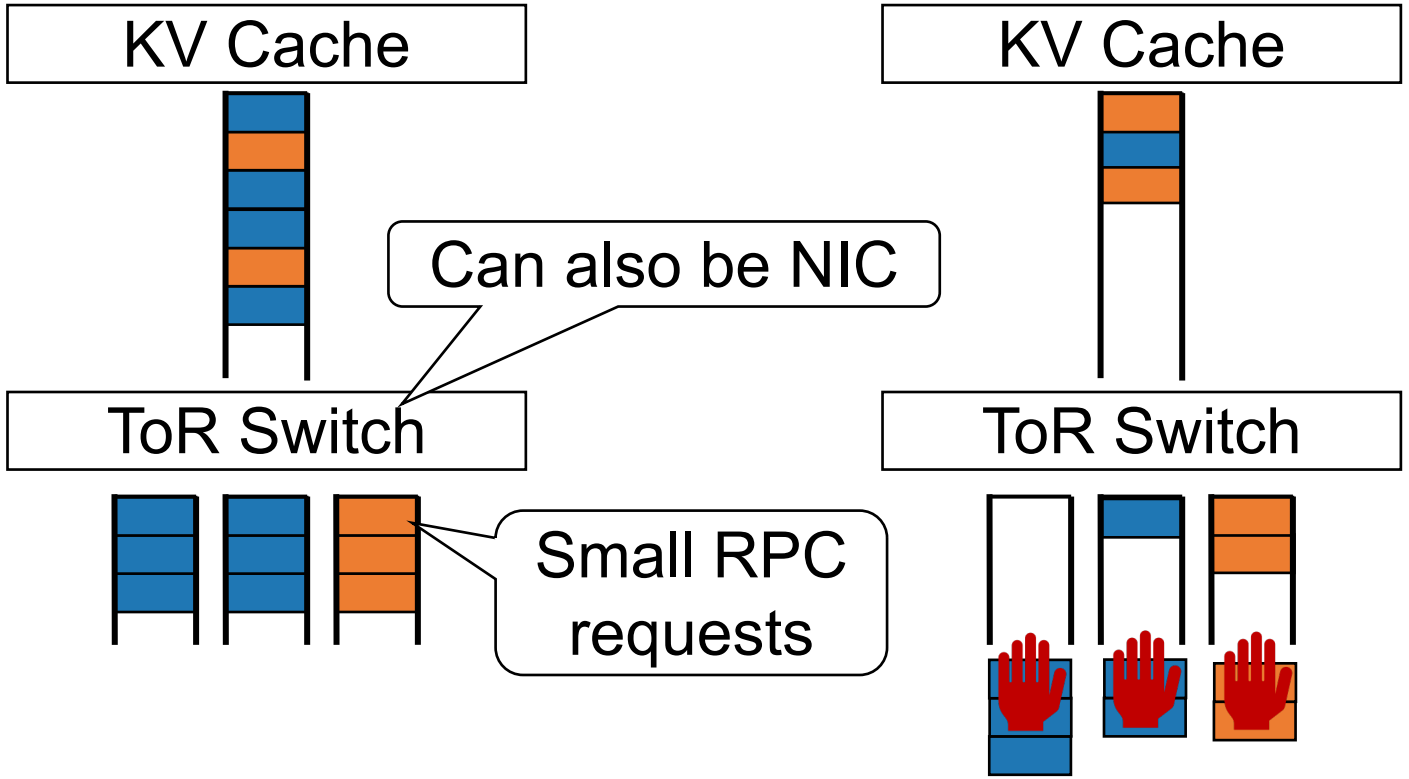
- These black boxes can be widely deployed.
- Local schedulers [e.g., PIFO (SIGCOMM ‘16)] do not apply.

Idea: schedule operations just before issuing to hardware!

Insight: Controlled queuing can favor an entity over another

Queue buildup triggers default queuing discipline

Workload composition can be changed when queues are empty



But, too empty queues cause starvation

Need to carefully match issued throughput with offload's throughput!

Challenge 2: unknown throughput

Offloads can have **different** and **varying** throughput due to

- complexity of logic
- hardware resources provisioned at a time

Black-box offloads cannot be instrumented to provide

- entity-level scheduling
- explicit feedback (throughput or congestion signals, e.g., ECN)

As such, isolation techniques that assume fixed throughput or rely on support from bottleneck do not apply

- e.g., Seawall (NSDI ‘11), FairCloud (SIGCOMM ‘12)

Idea: probe for throughput!

Probing for throughput

Strawman: observing throughput completed by offload

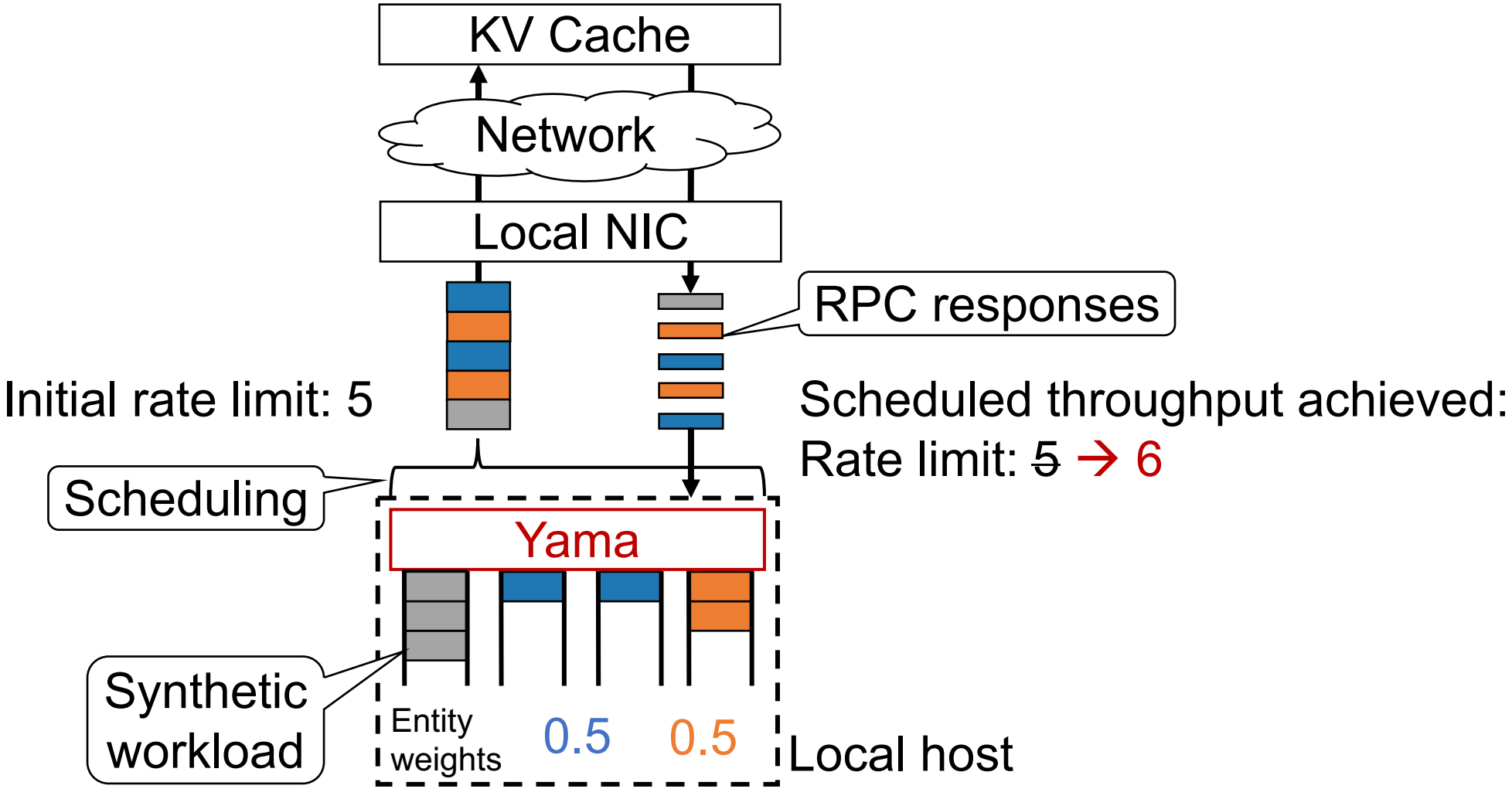
- Underestimation if applications don't have enough workload to saturate offload

Idea: "backfilling" – generating just enough synthetic workload to saturate offload

- offload throughput = application + synthetic workload throughput
- careful not to hurt application throughput

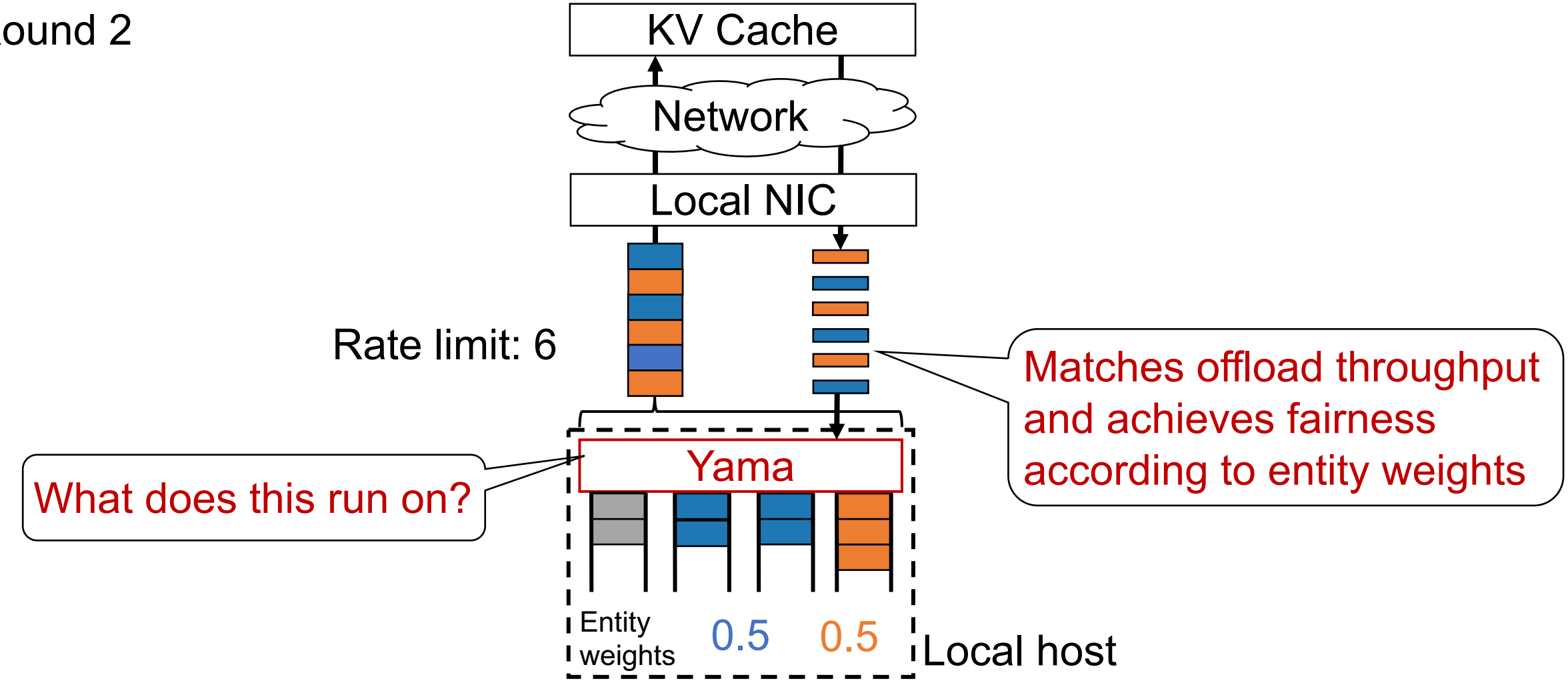
Putting scheduling and probing together

Round 1



Putting scheduling and probing together

Round 2



Executing scheduling and probing routines

Strawman: using dedicated CPU core(s)

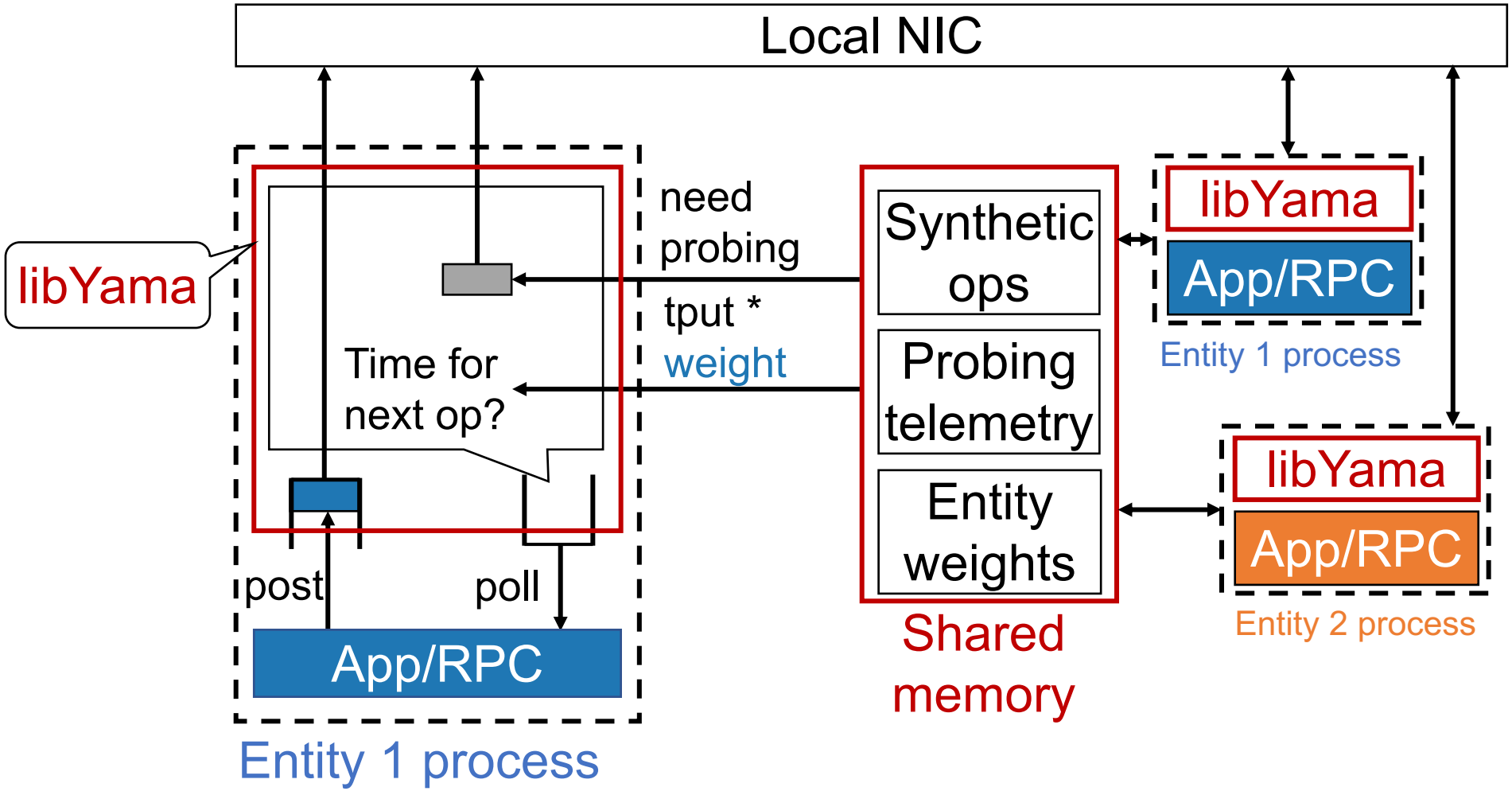
- Heavy resource overhead
- Hard to keep up with bandwidth scaling

Insight: applications spend many cycles busy polling for events such as RPC responses and RDMA CQEs.

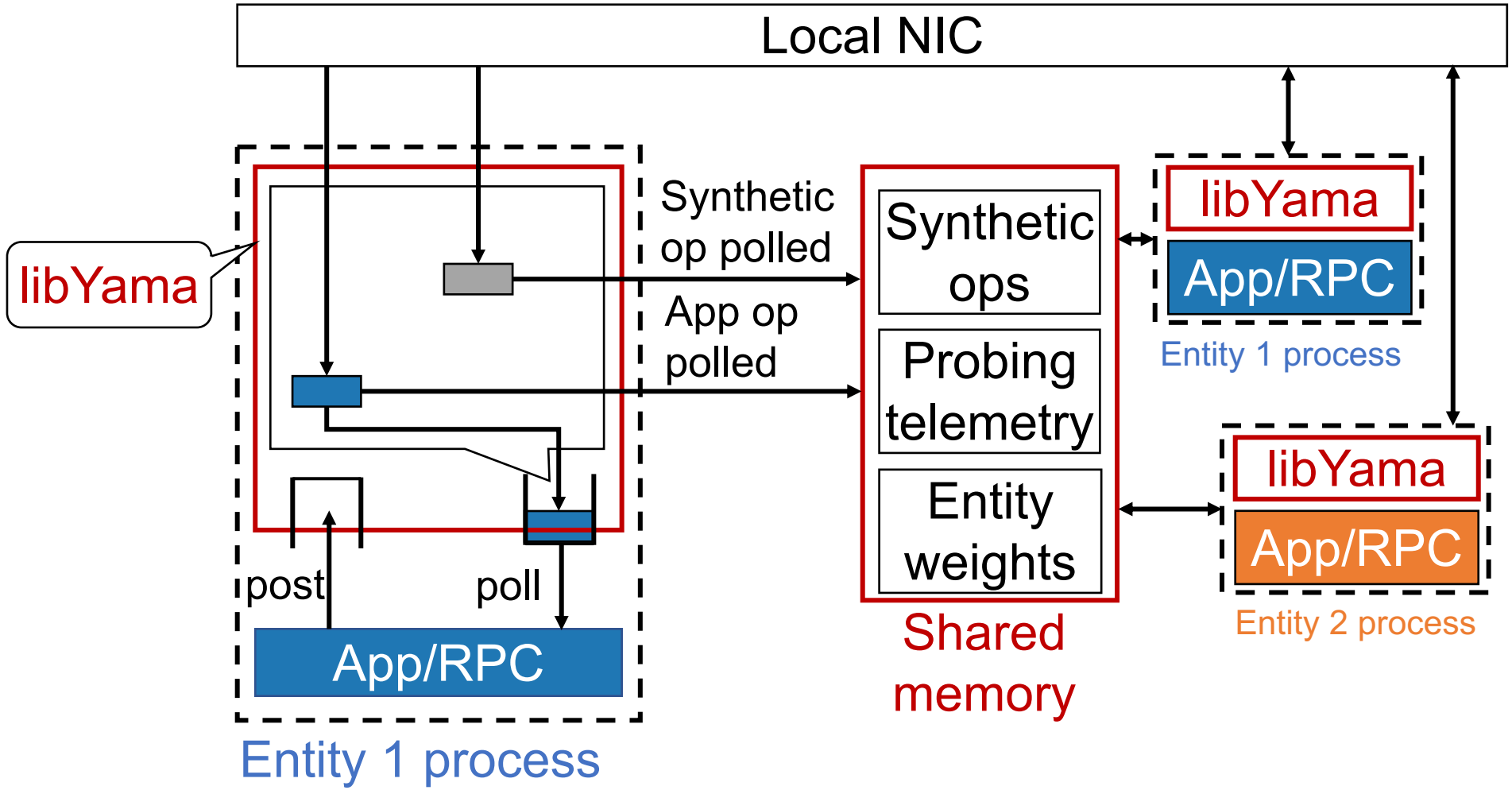
- These cycles would be “wasted” if we don’t leverage them

Idea: “cycle scavenging” – executing Yama scheduling and probing routines with application busy-polling cycles!

Cycle scavenging with `libYama`



Cycle scavenging with `libYama`



Other problems

Offload can be shared by multiple remote nodes!

- Elect a leader node to execute probing routine
- Followers exchange telemetry and probed throughput with leader

Offloads can form chains!

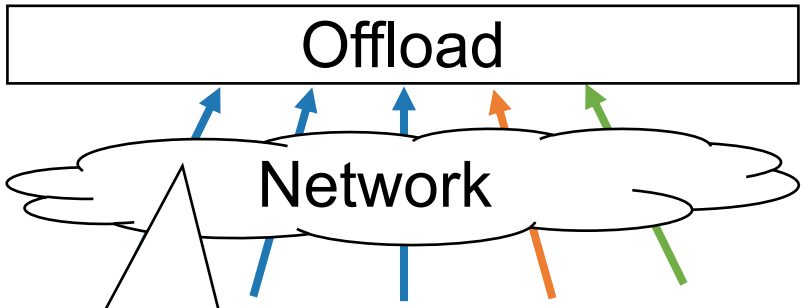
- We treat an offload chain as a black-box offload
- Complication: chains share bottleneck offload
  - Probing-scheduling feedback loop makes sure bottleneck is equally shared by chains

See our paper for more details

Evaluation overview

1. Can Yama achieve weighted sharing of offload?
2. Can Yama fairly share common bottleneck of two chains?
3. Does cycle scavenging significantly slow down app ops?

Q1: Can Yama achieve weighted sharing of offload?



One offload at each level:

- KV Cache: L7
- RDMA: L4
- NAT: L3

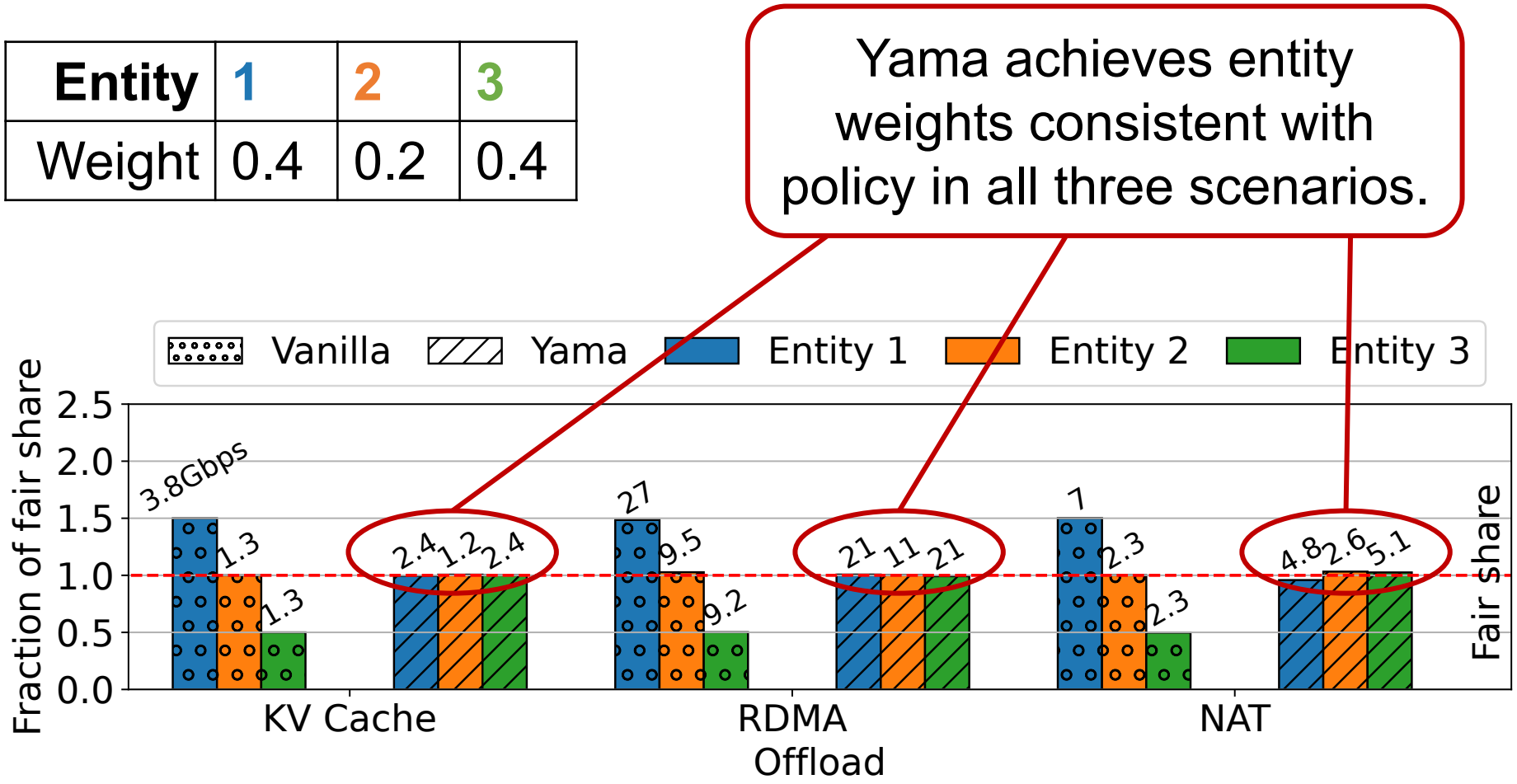
Policy

Entity	1	2	3
Weight	0.4	0.2	0.4
# IO queues	60%	20%	20%

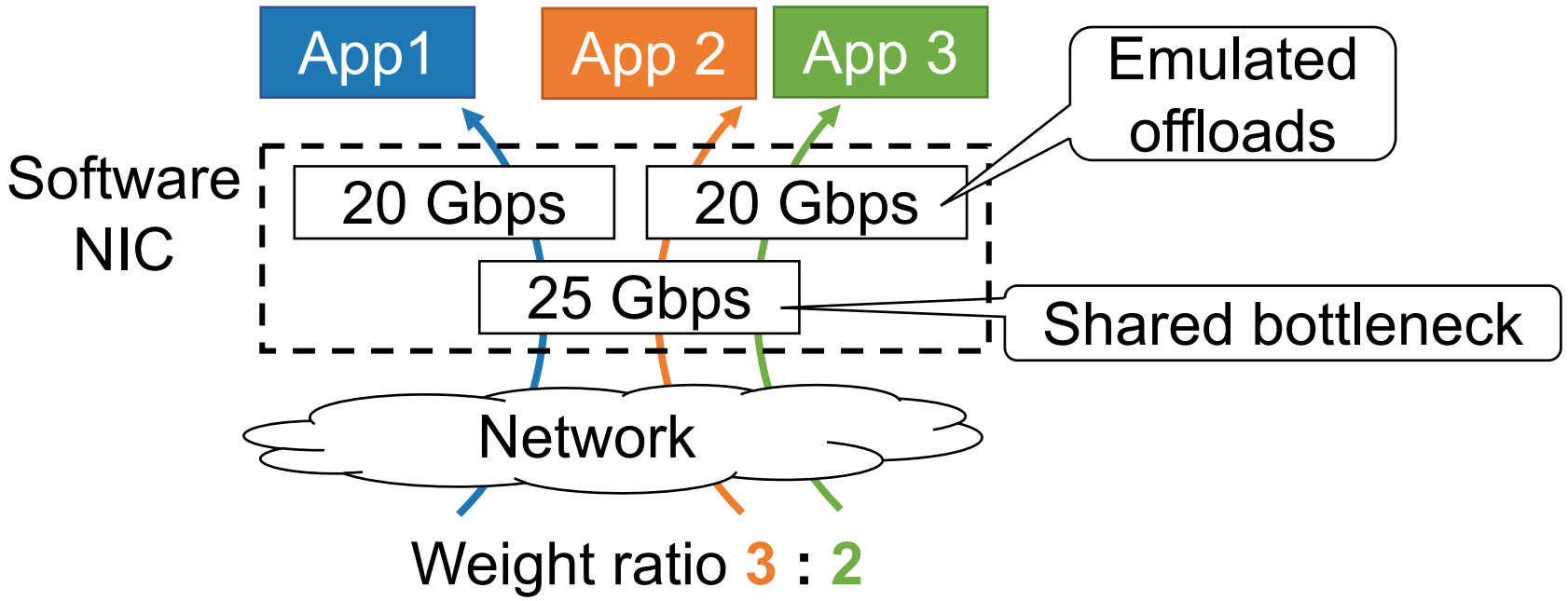
Yama should allocate, compared to default queuing discipline, less throughput to Entity 1, despite more queues, and vice versa for Entity 3.

Q1: Can Yama achieve weighted sharing of offload?

Entity	1	2	3
Weight	0.4	0.2	0.4



Q2: Can Yama fairly share common bottleneck of two chains?



Q2: Can Yama fairly share common bottleneck of two chains?

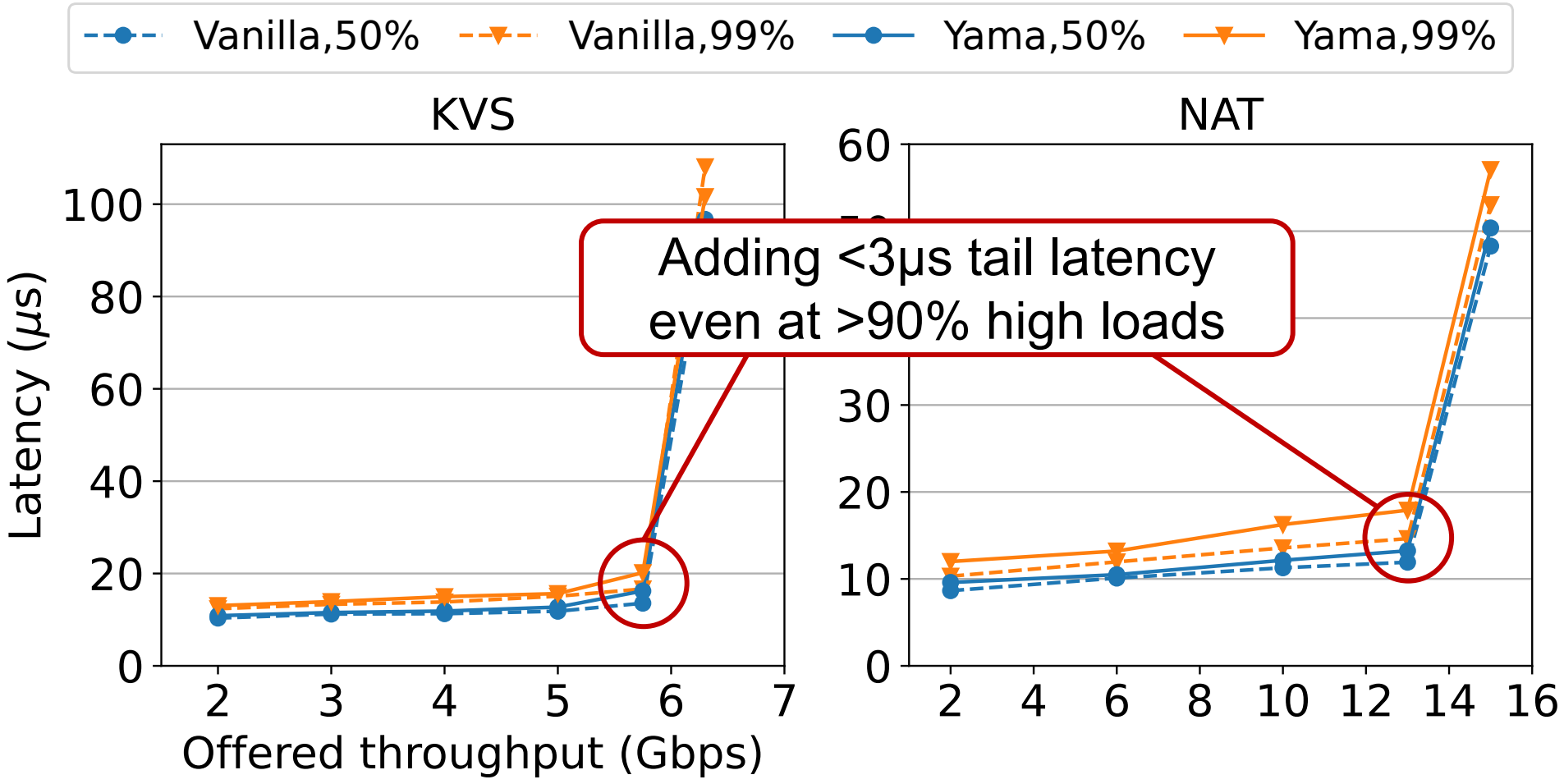
Synthetic ops don't hurt application throughput

App	Vanilla		Yama	
	Tput (Gbps)	%	Tput (Gbps)	%
(Total)	22.31	100	23.79	100
1	7.54	33.8	11.66	49.0
2	7.39	33.1	7.33	30.8
3	7.38	33.1	4.80	20.2

Yama fairly splits throughput between two chains.

Weighted fairness achieved within a chain.

Q3: Does cycle scavenging significantly slow down app ops?



Conclusions

- Entity-level performance isolation for black-box offloads is desired
- Yama uses synthetic ops to probe for bottleneck offload throughput and based on it schedules application ops.
- Yama scavenges application busy-poll cycles to run probing and scheduling routines.
- Yama achieves entity-level fair sharing of bottleneck throughput for individual offloads and offload chains with low overhead.

Thank you!