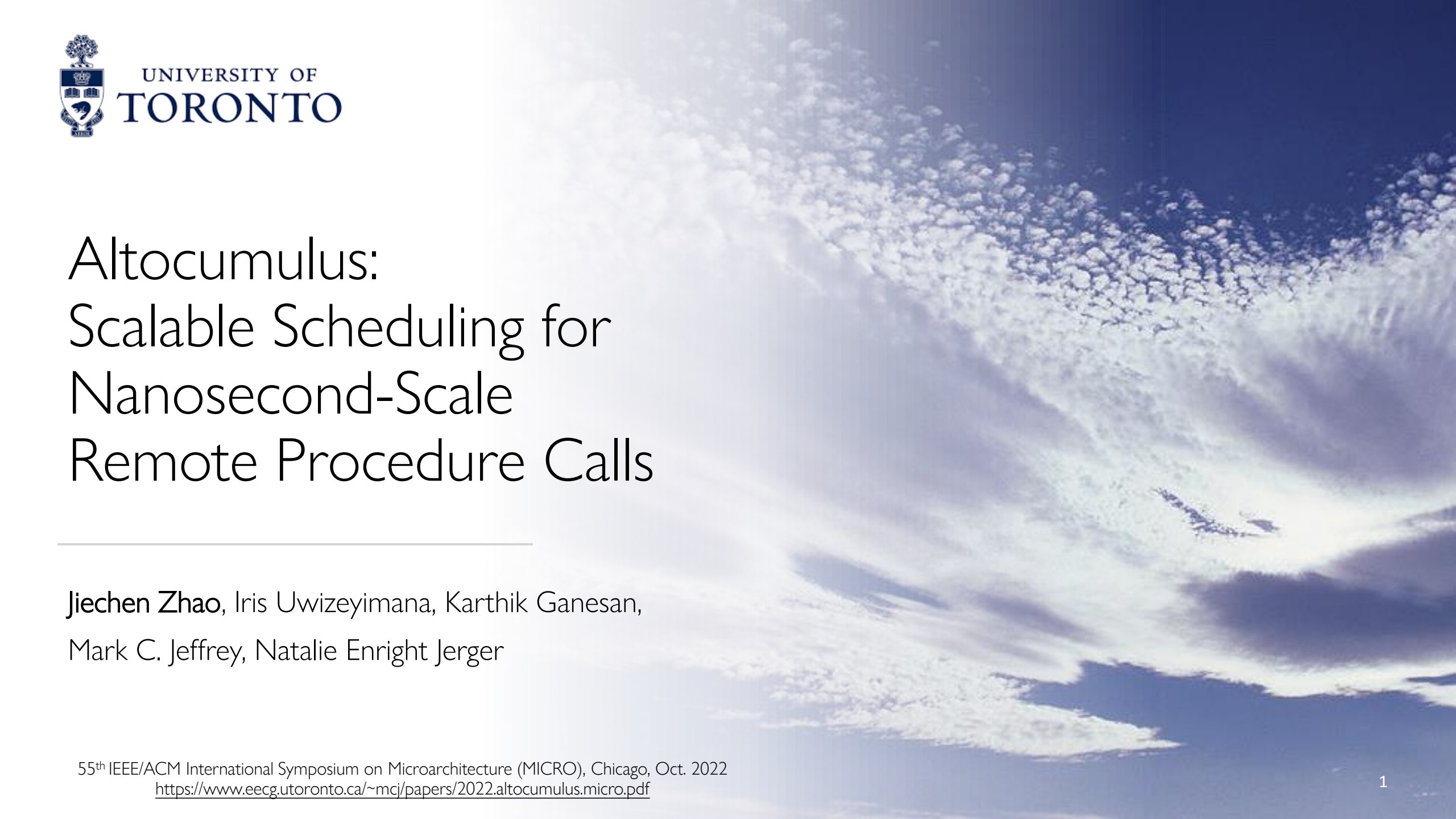


UNIVERSITY OF TORONTO

Altocumulus: Scalable Scheduling for Nanosecond-Scale Remote Procedure Calls

Jiechen Zhao, Iris Uwizeyimana, Karthik Ganesan,
Mark C. Jeffrey, Natalie Enright Jerger

55th IEEE/ACM International Symposium on Microarchitecture (MICRO), Chicago, Oct. 2022
https://www.eecg.utoronto.ca/~mcj/papers/2022.altocumulus.micro.pdf

Executive Summary

- RPC processing time has decreased down to microsecond scale
  - Therefore, bottleneck has shifted to scheduling
- Prior RPC schedulers react to imbalance ineffectively, causing
  - High tail latency at medium load
  - Low CPU utilization while satisfying strict tail latency deadline
- **Altocumulus:** Proactively rebalances RPC loads using HW/SW co-design
- Achieves up to 24x throughput improvement under microsecond scale tail latency deadline over state-of-the art

RPCs become ubiquitous in datacenters

- Productivity improvement
  - As a common communication API
  - Harmonizing distributed services developed by different programming languages
- Vast deployment
  - As the backbone of many latency-critical applications (KVS, SMR, RDMA, etc.)
- Massive amount
  - Each request involving 10s - 1,000s RPCs

APACHE THRIFT

gRPC

RPC Handling – Perf. Critical !

RPC Performance is Critical

- Many datacenter applications today are interactive
  - Strict performance requirements as *SLO*: Low tail latency at high load

RPC Performance is Critical

- Many datacenter applications today are interactive
  - Strict performance requirements as *SLO*: Low tail latency at high load
- "Tail-at-scale" problem [Dean et al., CACM'13]
  - User-perceived latency determined by slowest back-end server node

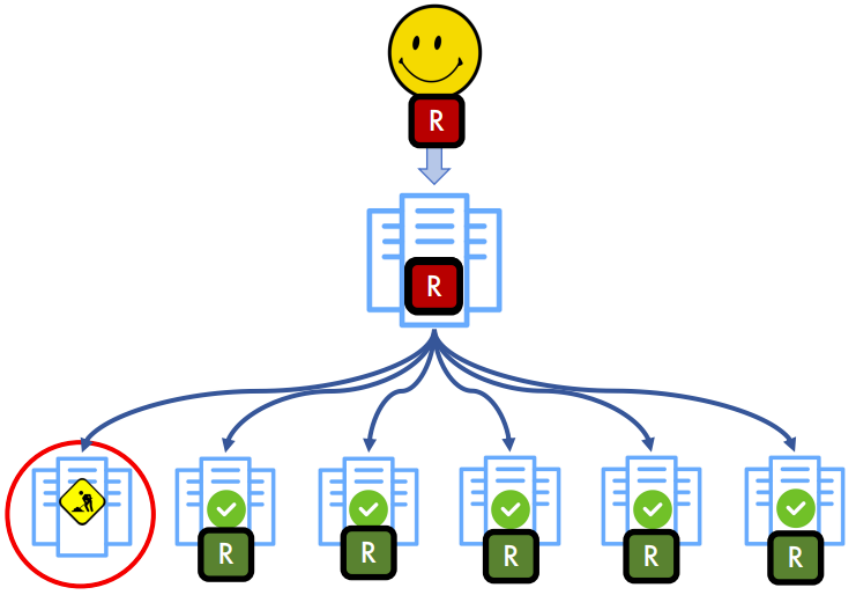


Fig. from [Kaffes et al., NSDI'19]

RPC Performance is Critical

- Many datacenter applications today are interactive
  - Strict performance requirements as *SLO*: Low tail latency at high load
- "Tail-at-scale" problem [Dean et al., CACM'13]
  - User-perceived latency determined by slowest back-end server node

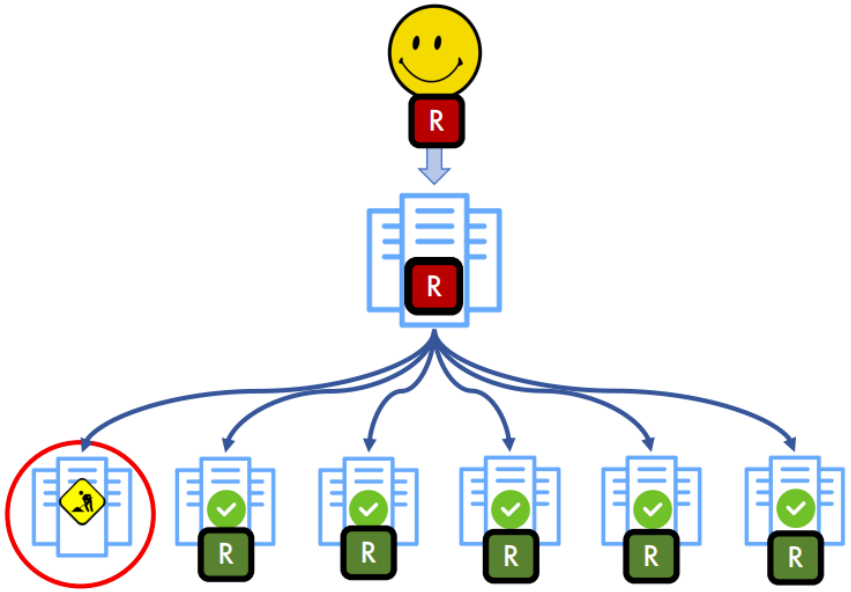


Fig. from [Kaffes et al., NSDI'19]

Why on-CPU RPC handling counts ?

RPC Performance is Critical

- Many datacenter applications today are interactive
  - Strict performance requirements as *SLO*: Low tail latency at high load
- "Tail-at-scale" problem [Dean et al., CACM'13]

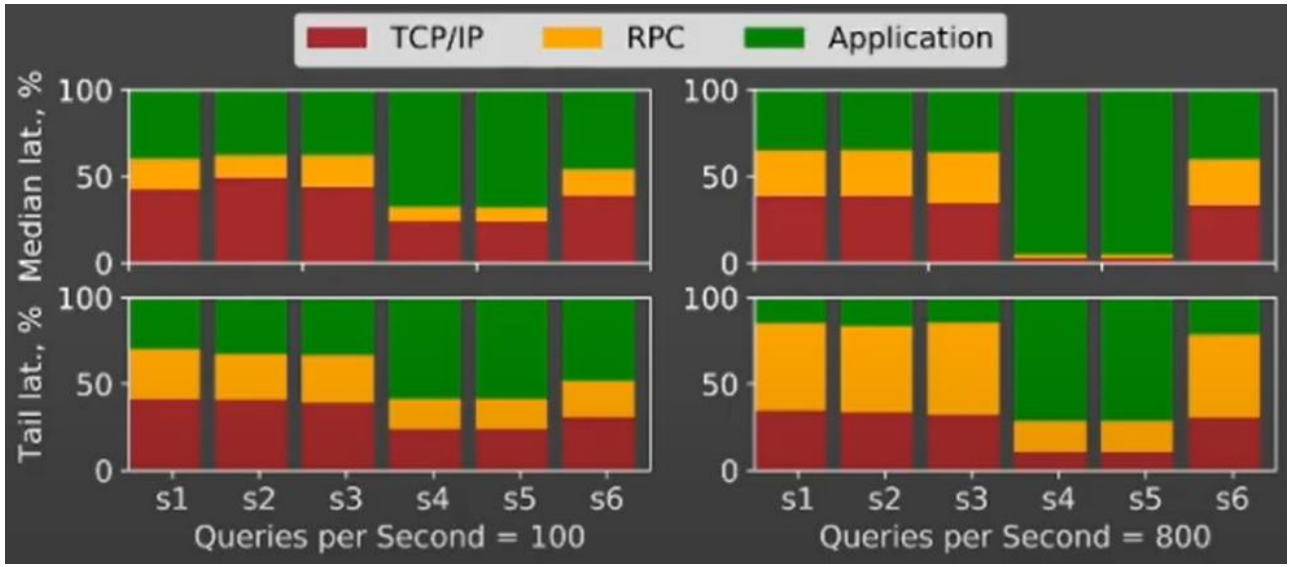


Fig. from [Lazarev et al., ASPLOS'21]

Why on-CPU RPC handling counts ?

RPC Performance is Critical

- Many datacenter applications today are interactive
  - Strict performance requirements as *SLO*: Low tail latency at high load
- "Tail-at-scale" problem [Dean et al., CACM'13]

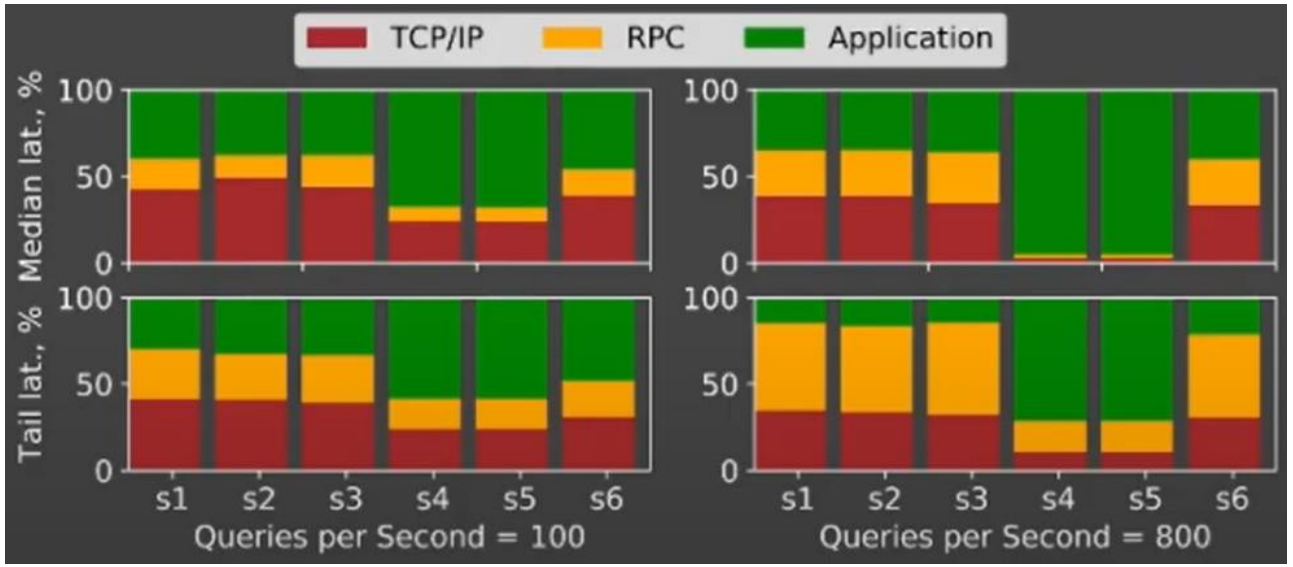


Fig. from [Lazarev et al., ASPLOS'21]

Why on-CPU RPC handling counts ?

Taking up to ~90% of end-to-end time

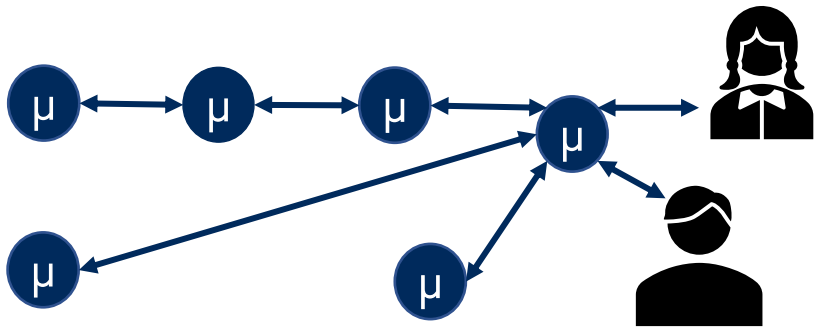
RPC Performance is Critical

- Many datacenter applications today are interactive
  - Strict performance requirements as *SLO*: Low tail latency at high load
- "Tail-at-scale" problem [Dean et al., CACM'13]
- "Killer microsecond" problem [Barroso et al., CACM'17]
  - Existing systems not able to handle microsecond-scale RPCs efficiently

What's worse …

RPC Performance is Critical

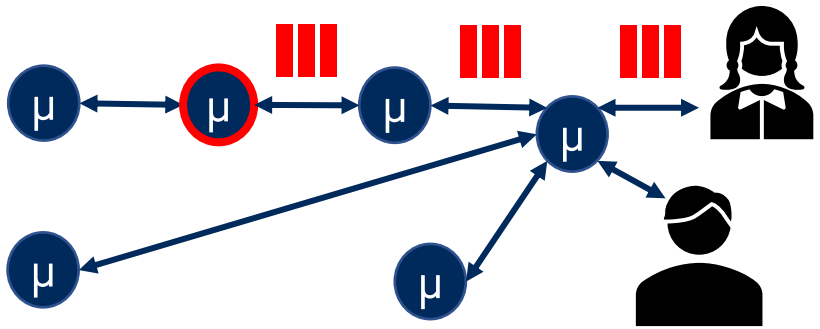
- Many datacenter applications today are interactive
  - Strict performance requirements as *SLO*: Low tail latency at high load
- "Tail-at-scale" problem [Dean et al., CACM'13]
- "Killer microsecond" problem [Barroso et al., CACM'17]
- Microservice queuing propagation
  - Each server receiving/sending RPCs whose delay varies



What's even worse …

RPC Performance is Critical

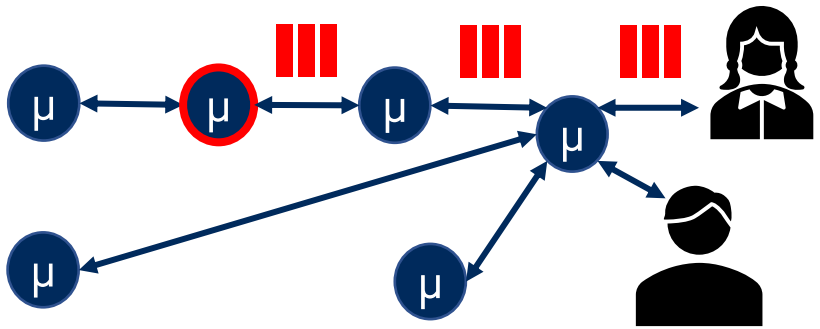
- Many datacenter applications today are interactive
  - Strict performance requirements as *SLO*: Low tail latency at high load
- "Tail-at-scale" problem [Dean et al., CACM'13]
- "Killer microsecond" problem [Barroso et al., CACM'17]
- Microservice queuing propagation
  - Each server receiving/sending RPCs whose delay varies



What's even worse …

RPC Performance is Critical

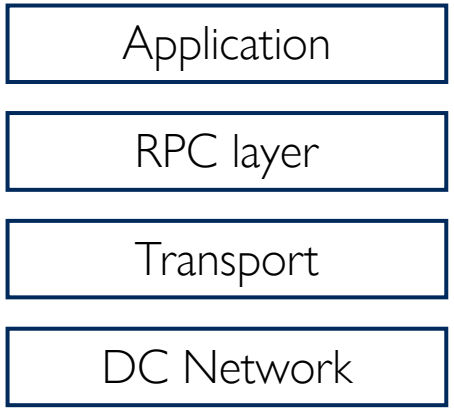
- Many datacenter applications today are interactive
  - Strict performance requirements as *SLO*: Low tail latency at high load
- "Tail-at-scale" problem [Dean et al., CACM'13]
- "Killer microsecond" problem [Barroso et al., CACM'17]
- Microservice queuing propagation
  - Each server receiving/sending RPCs whose delay varies



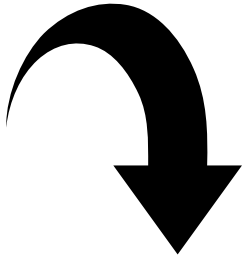
On-CPU RPC handling SLO
~= 10s microsecond

RPC Lifetime on a CPU

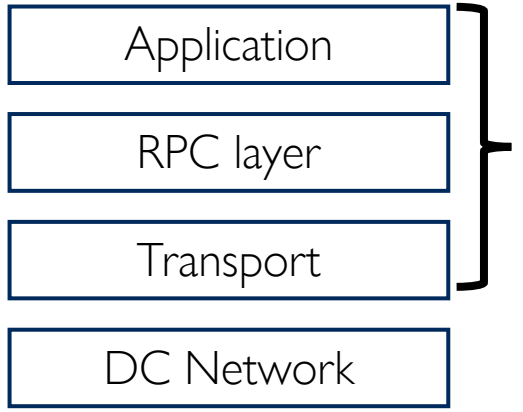
- Processing



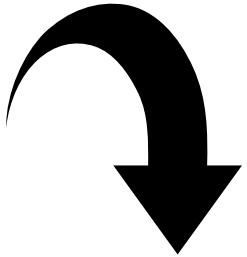
RPC Lifetime on a CPU



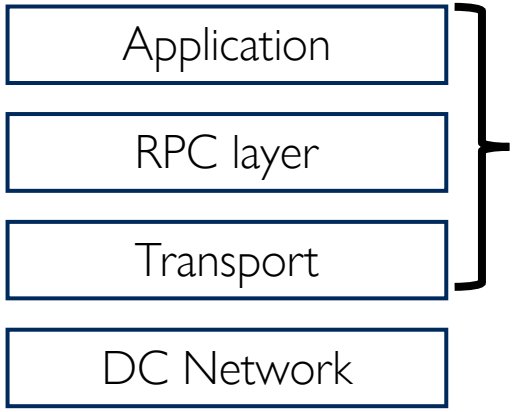
- Processing



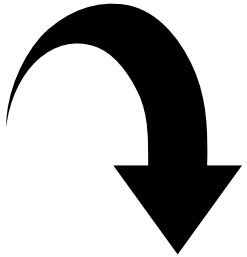
RPC Lifetime on a CPU



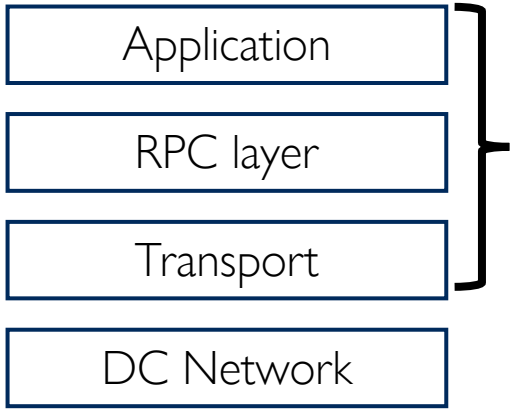
- Processing
  - Application
    - monolithic ⟶ microservices (10s ns – 10s µs)



RPC Lifetime on a CPU

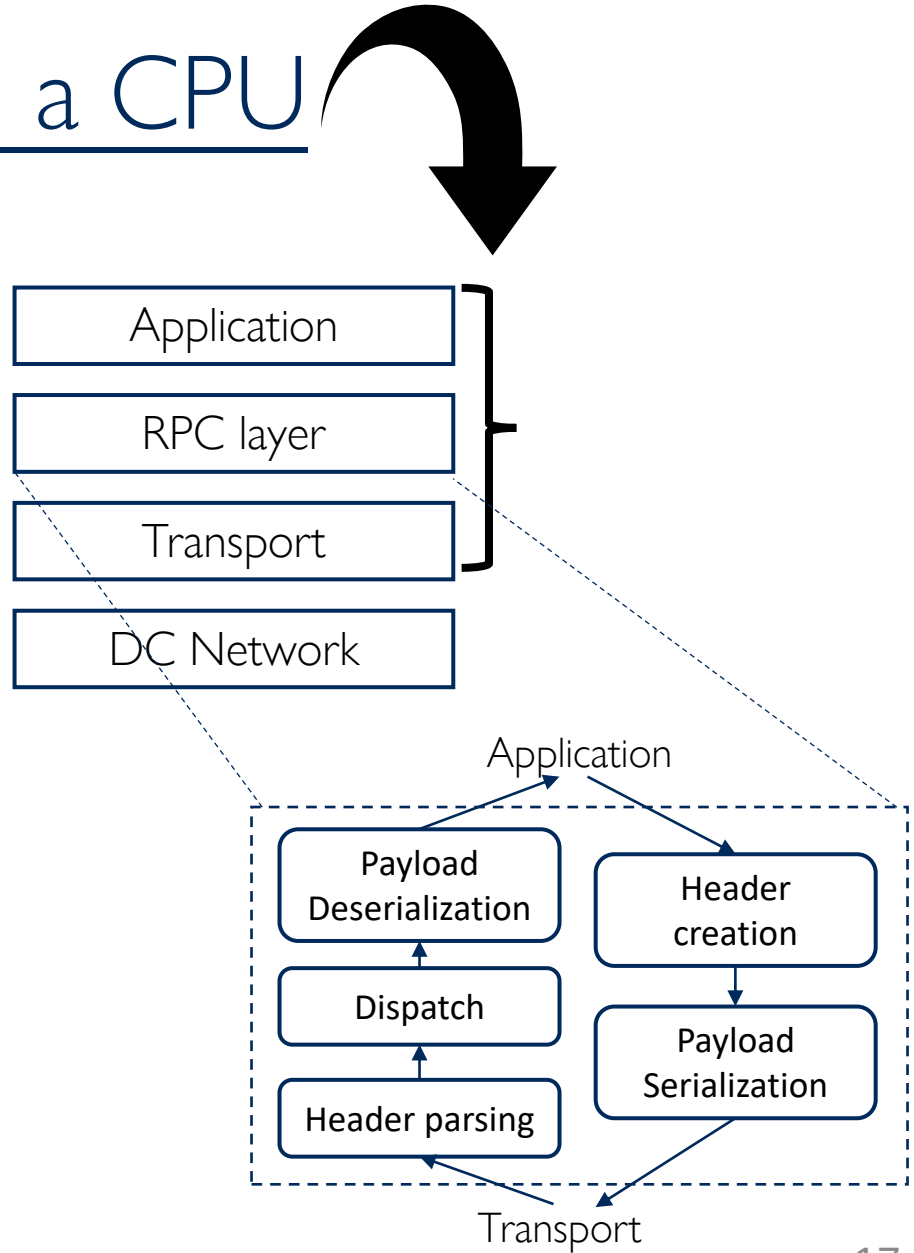


- Processing
  - Application
    - monolithic ⟶ microservices (10s ns – 10s µs)
  - RPC layer
    - Software ⟶ hardware (10s ns)

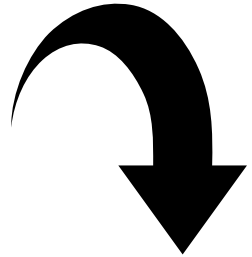


RPC Lifetime on a CPU

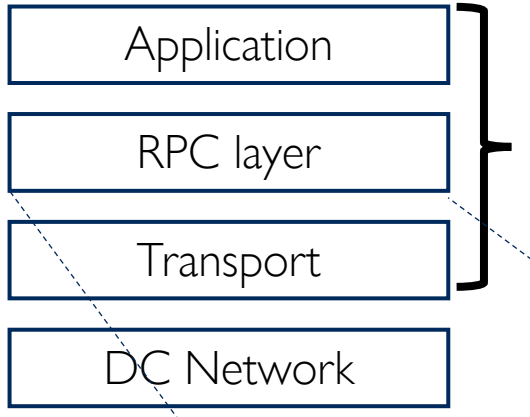
- Processing
  - Application
    - monolithic ⟶ microservices (10s ns – 10s µs)
  - RPC layer
    - Software ⟶ hardware (10s ns)



RPC Lifetime on a CPU



- Processing
  - Application
    - monolithic ⟶ microservices (10s ns – 10s µs)
  - RPC layer
    - Software ⟶ hardware (10s ns)



Prior art on accelerating RPC layer

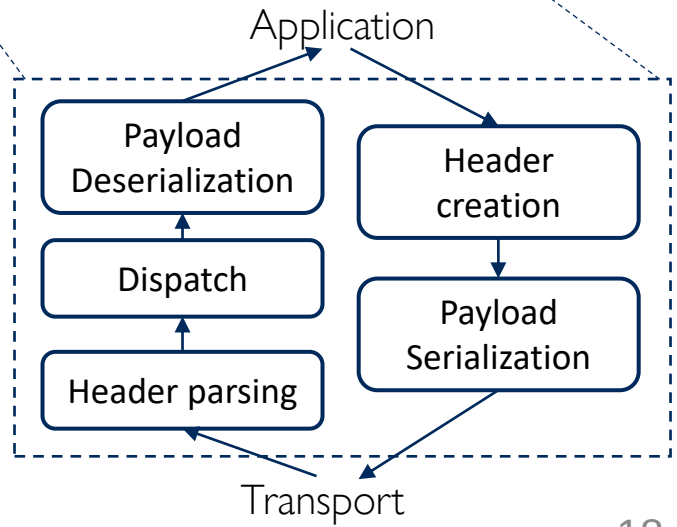
[Pourhabibi et al., ASPLOS'20]

[Pourhabibi et al., MICRO'21]

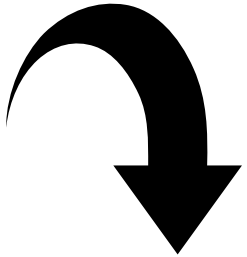
[Karandikar et al., MICRO'21]

[Lazarev et al., ASPLOS'21]

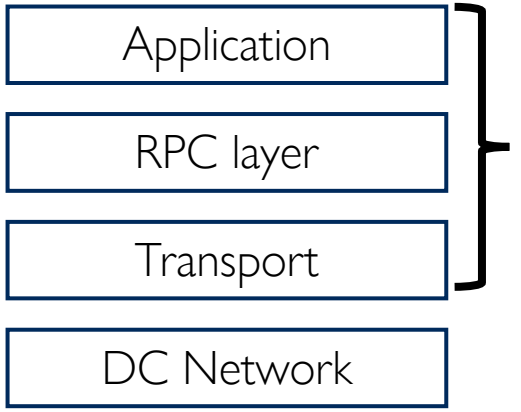
[Wolnikowski et al., HotOS'21]



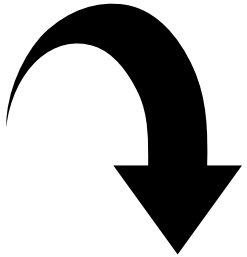
RPC Lifetime on a CPU



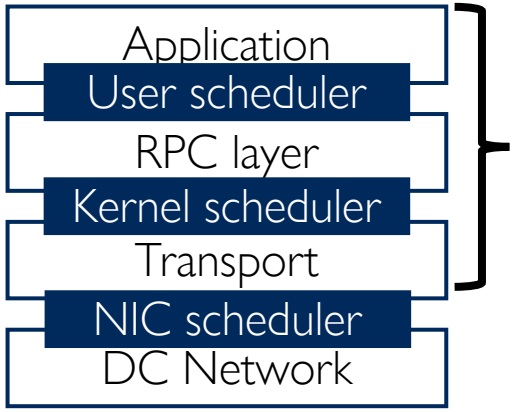
- Processing
  - Application
    - monolithic ⟶ microservices (10s ns – 10s µs)
  - RPC layer
    - Software ⟶ hardware (10s ns)
  - Transport
    - TCP/IP ⟶ optimization (100s ns)



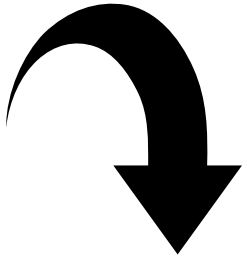
RPC Lifetime on a CPU



- Processing
  - Application
    - monolithic ⟶ microservices (10s ns – 10s µs)
  - RPC layer
    - Software ⟶ hardware (10s ns)
  - Transport
    - TCP/IP ⟶ optimization (100s ns)
- Scheduling ?



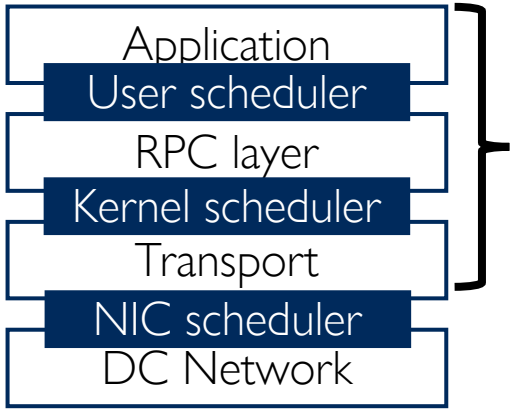
RPC Lifetime on a CPU



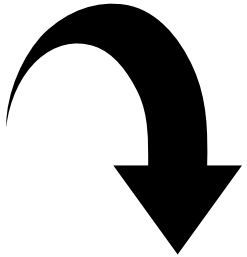
- Processing

Nanosecond scale

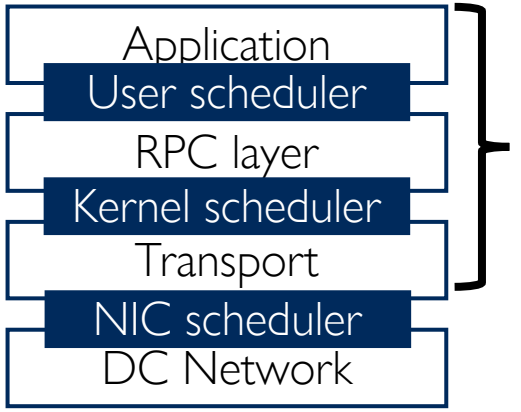
  - Application
    - monolithic ⟶ microservices (10s ns – 10s µs)
  - RPC layer
    - Software ⟶ hardware (10s ns)
  - Transport
    - TCP/IP ⟶ optimization (100s ns)
- Scheduling ?



RPC Lifetime on a CPU

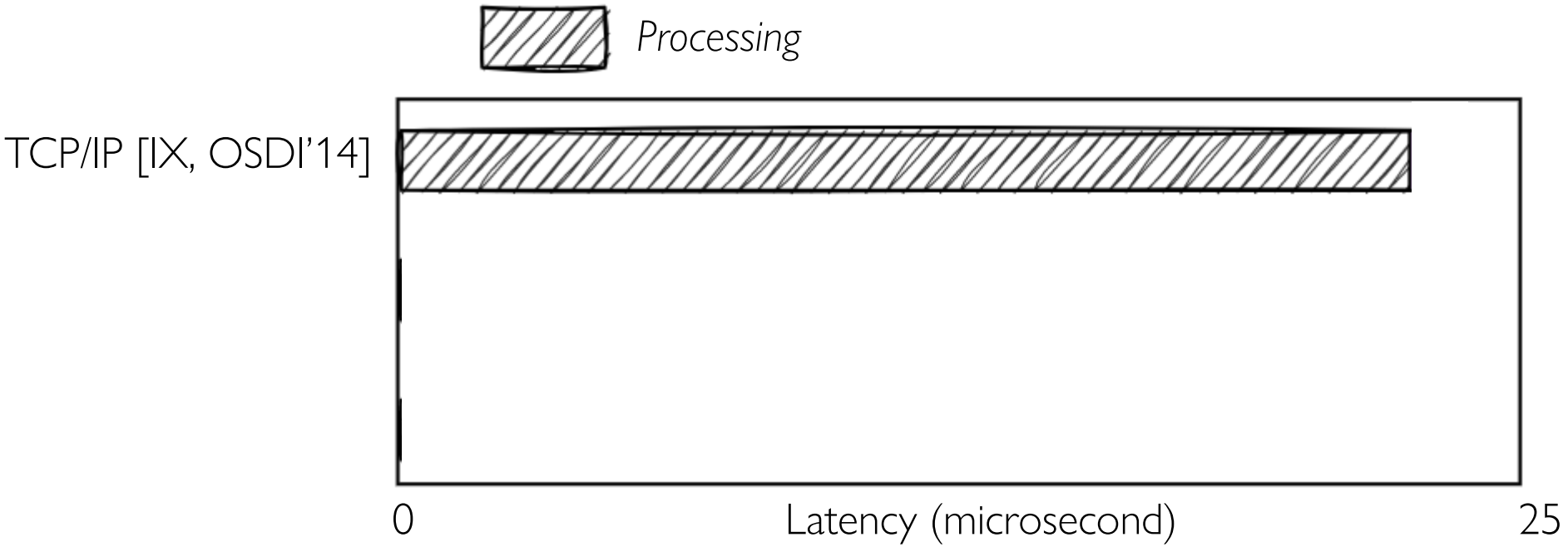


- Processing — Now, Nanosecond scale
  - Application
    - monolithic ⟶ microservices (10s ns – 10s µs)
  - RPC layer
    - Software ⟶ hardware (10s ns)
  - Transport
    - TCP/IP ⟶ optimization (100s ns)
- Scheduling ? — Now, impact on SLO ?



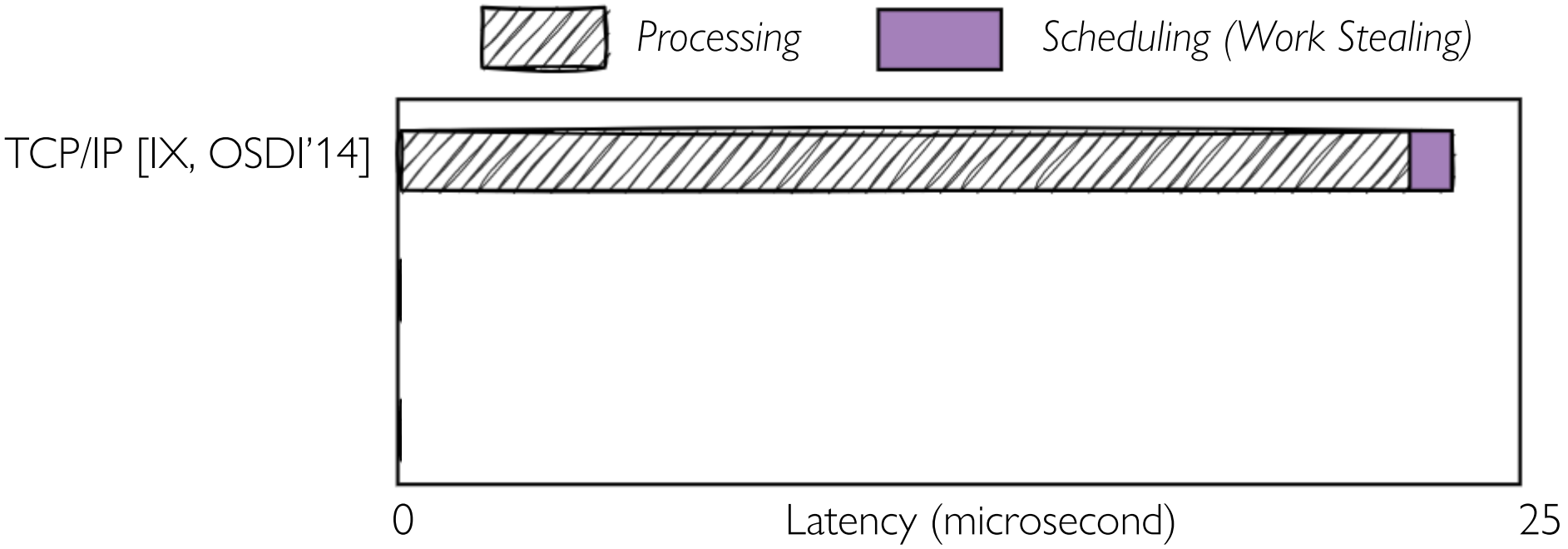
RPC Lifetime on a CPU: New Trends

- With prior work reducing the processing latency of RPCs …



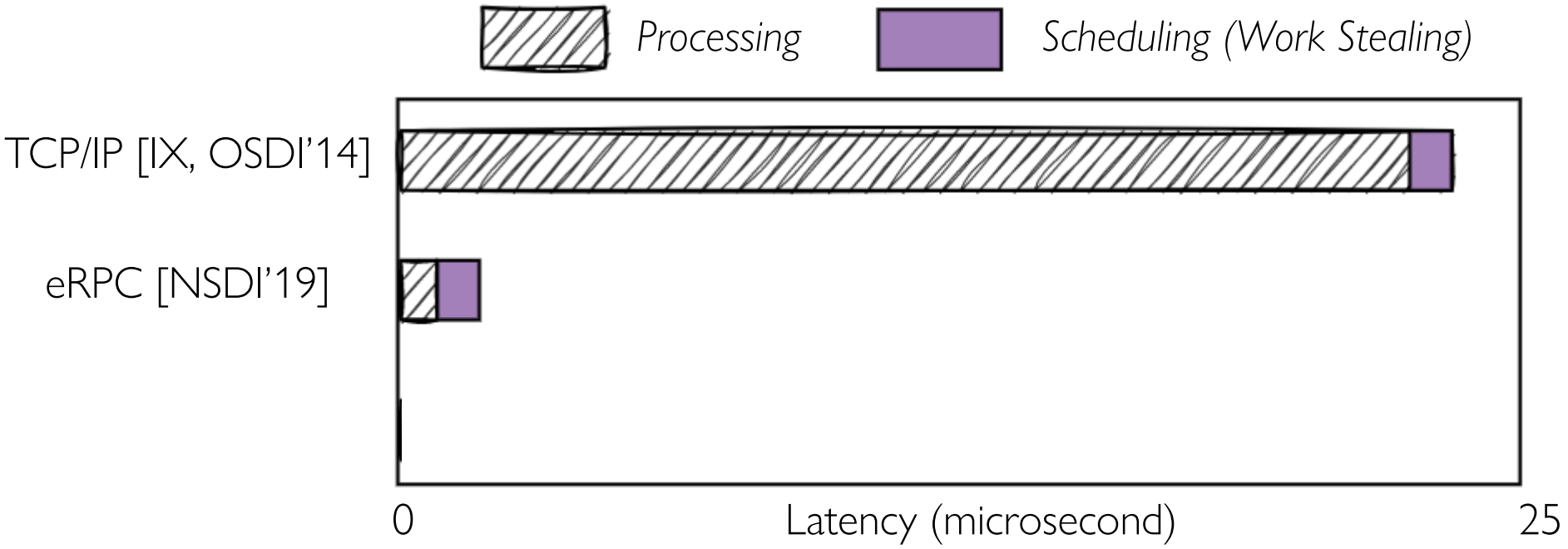
RPC Lifetime on a CPU: New Trends

- With prior work reducing the processing latency of RPCs …



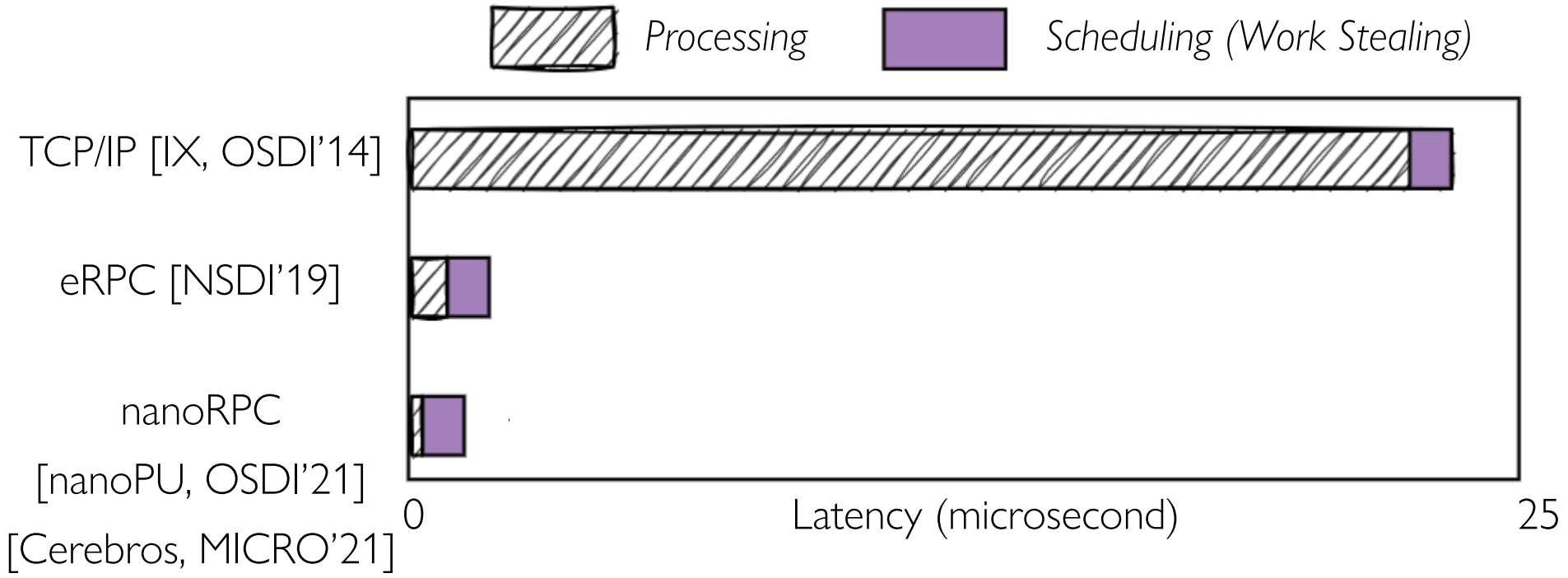
RPC Lifetime on a CPU: New Trends

- With prior work reducing the processing latency of RPCs …



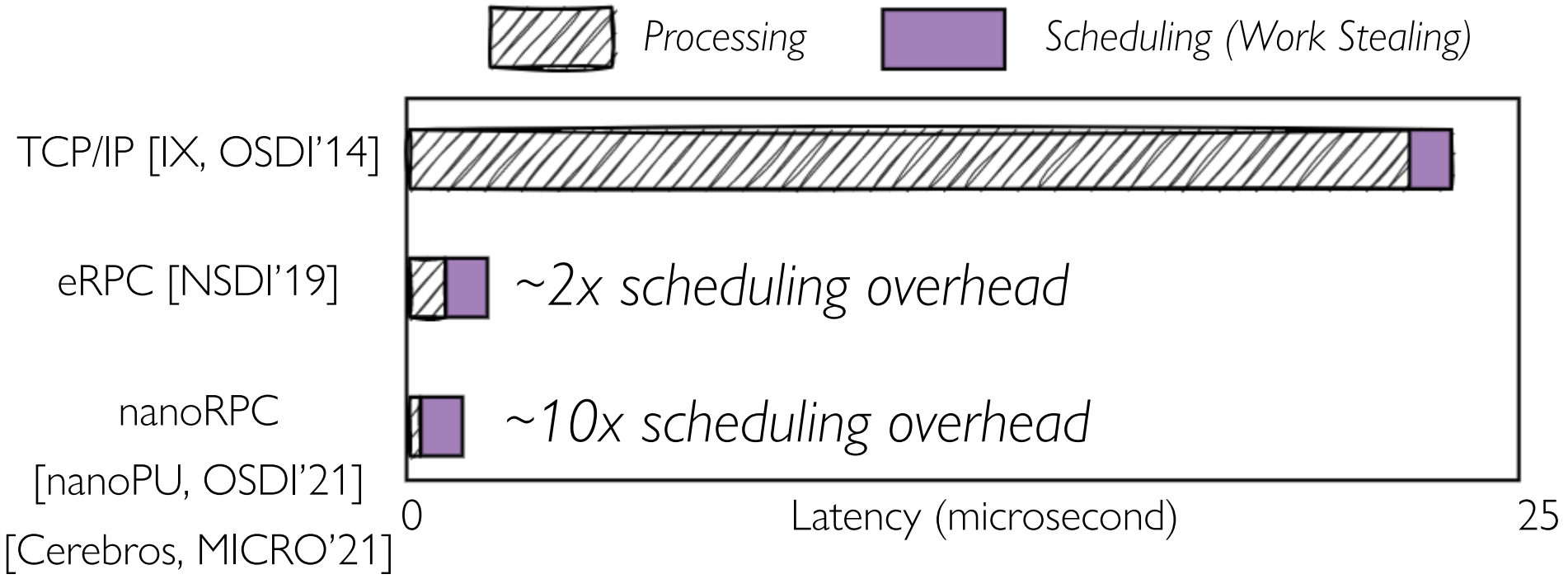
RPC Lifetime on a CPU: New Trends

- With prior work reducing the processing latency of RPCs …



RPC Lifetime on a CPU: New Trends

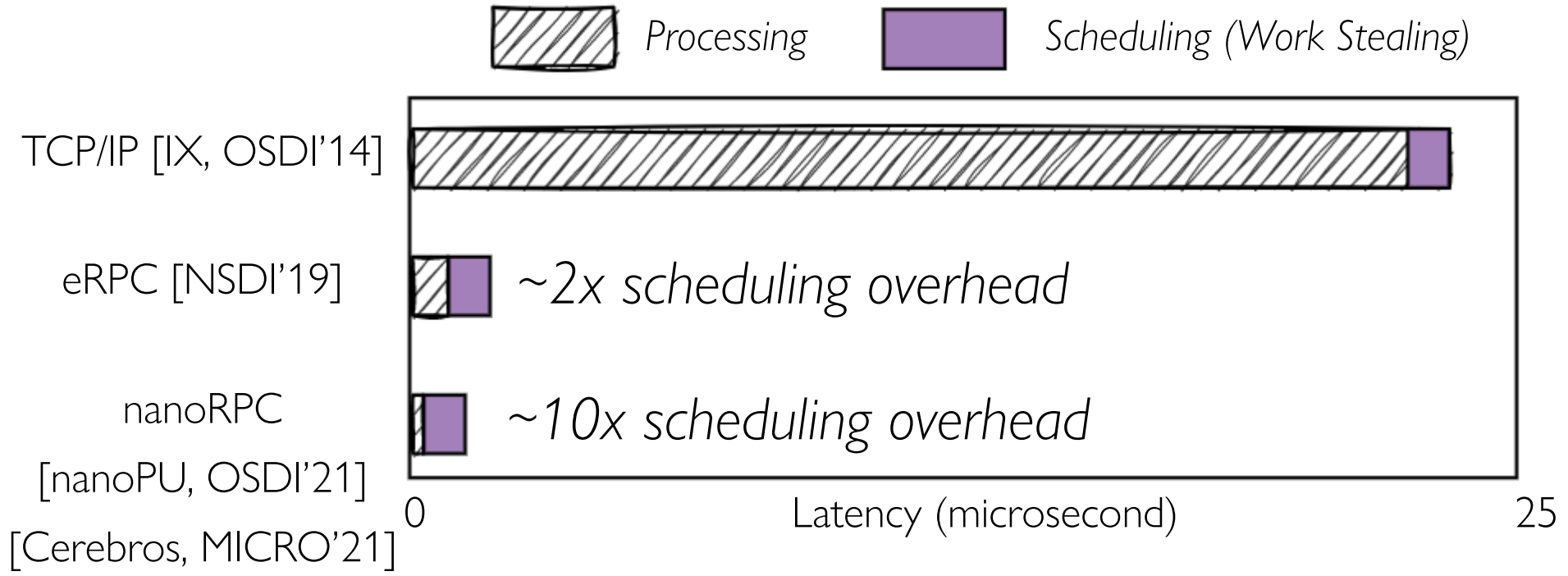
- With prior work reducing the processing latency of RPCs …



RPC Lifetime on a CPU: New Trends

- With prior work reducing the processing latency of RPCs …

Performance bottleneck shifted from RPC processing to RPC *scheduling*



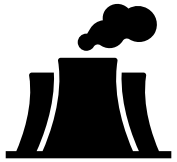
Key Questions

How fast should RPC scheduling be?

Minimize SLO violations

How many RPCs a server can sustain w/o violating SLO ?

Maximize RPC throughput@SLO



How well cores can be utilized ?

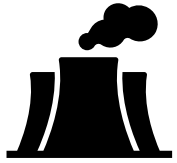
Be scalable to manycore CPUs (16 – 256 cores)

Design Goals

Reconcile the trade-off among

High volume of requests
(400+ Gbps)

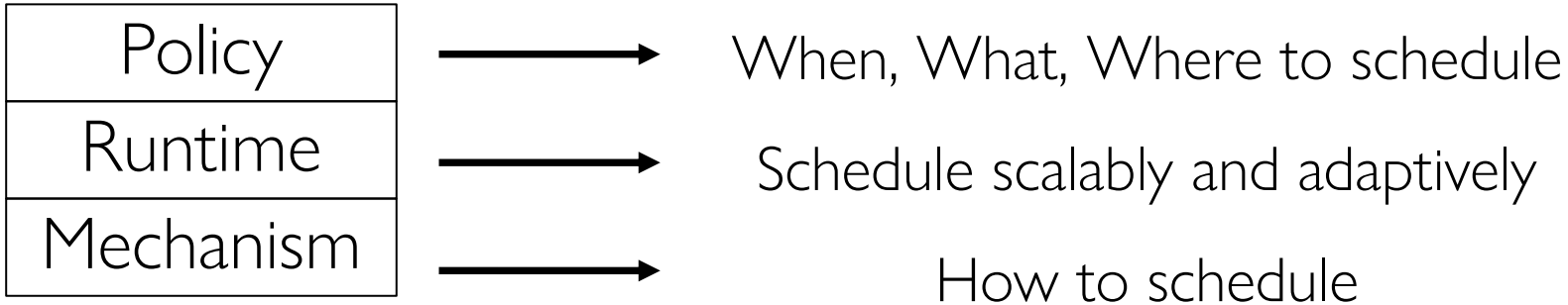
Strict SLO
(99th% latency@µs)



High CPU efficiency
(16-256 cores)

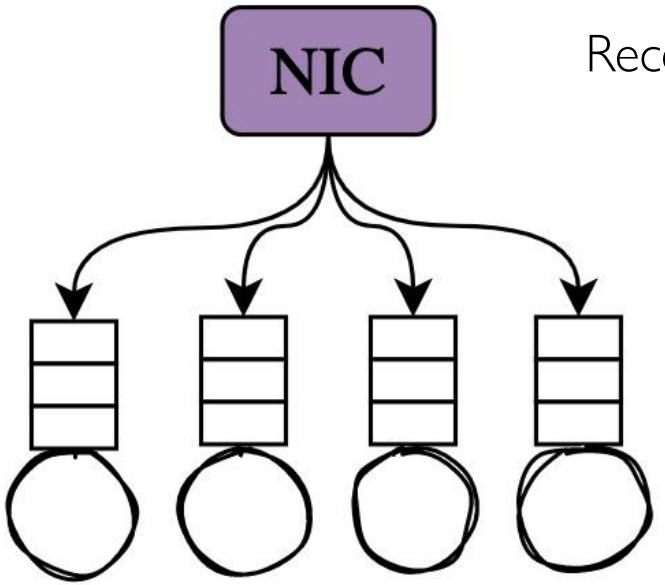
Technical design goal: <50ns scheduling overhead --- near-ideal !

Abstracting RPC Scheduling Sub-System



Bottlenecks in Prior RPC Schedulers

- Decentralized First Come First Serve (D-FCFS)



Receive Side Scaling (RSS)

Queues

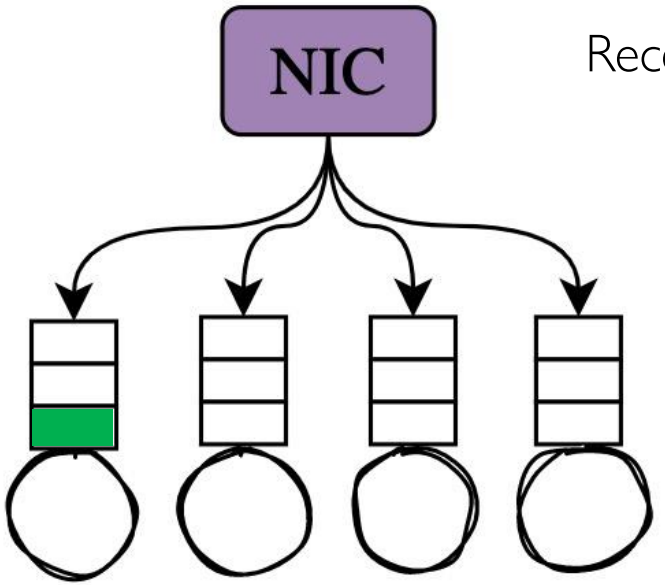
Cores

Kernel-native
d-FCFS runtime
Shared mem.

IX
[Belay et al., OSDI'14]

Bottlenecks in Prior RPC Schedulers

- Decentralized First Come First Serve (D-FCFS)



Receive Side Scaling (RSS)

Queues

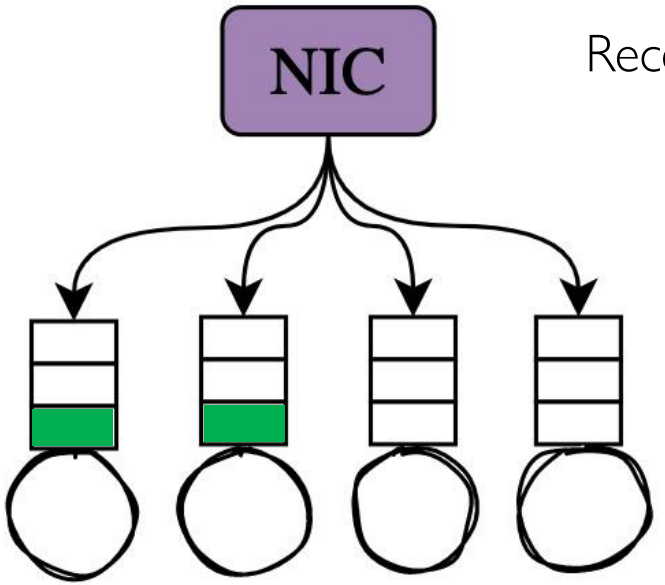
Cores

Kernel-native
d-FCFS runtime
Shared mem.

IX
[Belay et al., OSDI'14]

Bottlenecks in Prior RPC Schedulers

- Decentralized First Come First Serve (D-FCFS)



Receive Side Scaling (RSS)

Queues

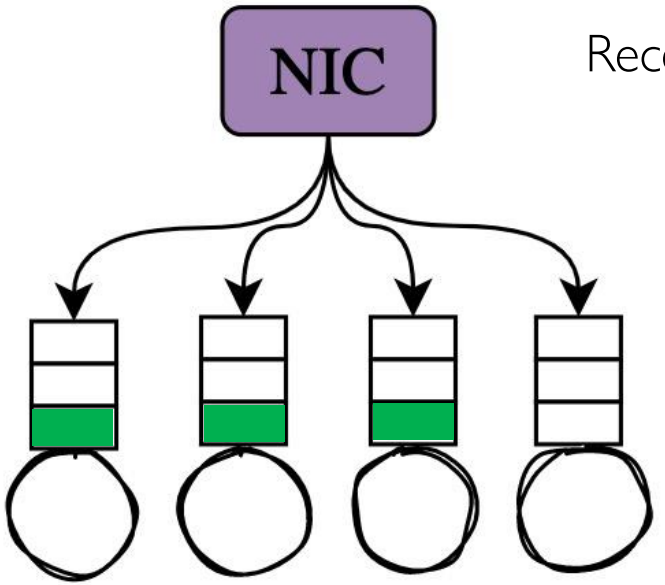
Cores

Kernel-native
d-FCFS runtime
Shared mem.

IX
[Belay et al., OSDI'14]

Bottlenecks in Prior RPC Schedulers

- Decentralized First Come First Serve (D-FCFS)



Receive Side Scaling (RSS)

Queues

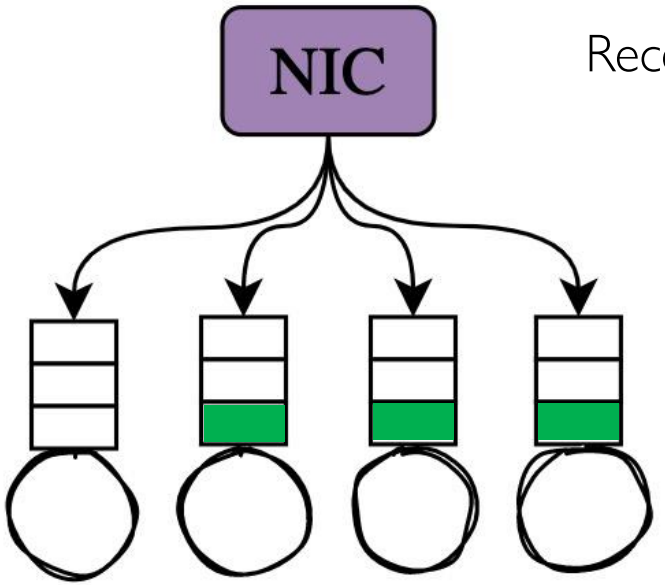
Cores

Kernel-native
d-FCFS runtime
Shared mem.

IX
[Belay et al., OSDI'14]

Bottlenecks in Prior RPC Schedulers

- Decentralized First Come First Serve (D-FCFS)



Receive Side Scaling (RSS)

Queues

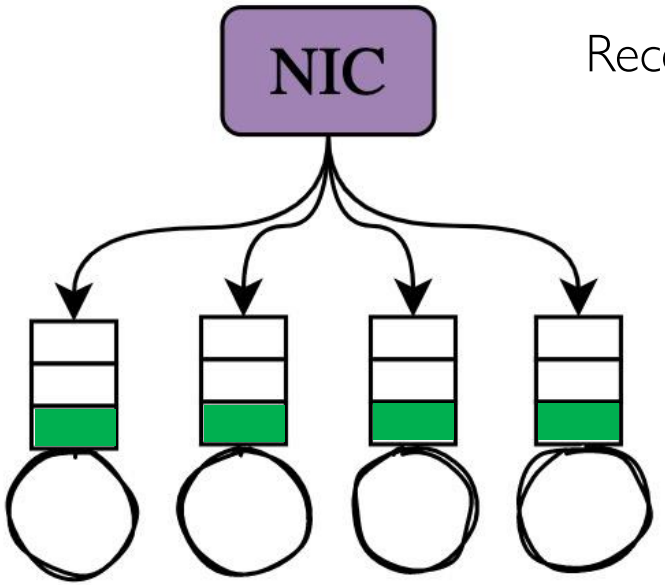
Cores

Kernel-native
d-FCFS runtime
Shared mem.

IX
[Belay et al., OSDI'14]

Bottlenecks in Prior RPC Schedulers

- Decentralized First Come First Serve (D-FCFS)



Receive Side Scaling (RSS)

Queues

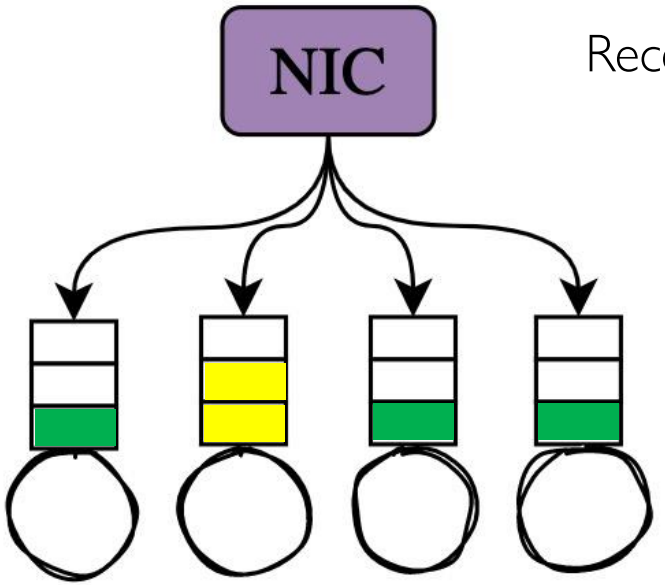
Cores

Kernel-native
d-FCFS runtime
Shared mem.

IX
[Belay et al., OSDI'14]

Bottlenecks in Prior RPC Schedulers

- Decentralized First Come First Serve (D-FCFS)



Receive Side Scaling (RSS)

Queues

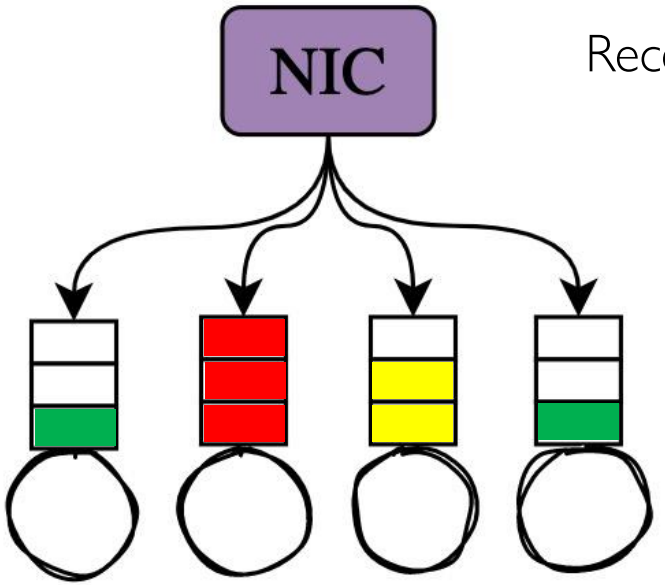
Cores

Kernel-native
d-FCFS runtime
Shared mem.

IX
[Belay et al., OSDI'14]

Bottlenecks in Prior RPC Schedulers

- Decentralized First Come First Serve (D-FCFS)



Receive Side Scaling (RSS)

Queues

Cores

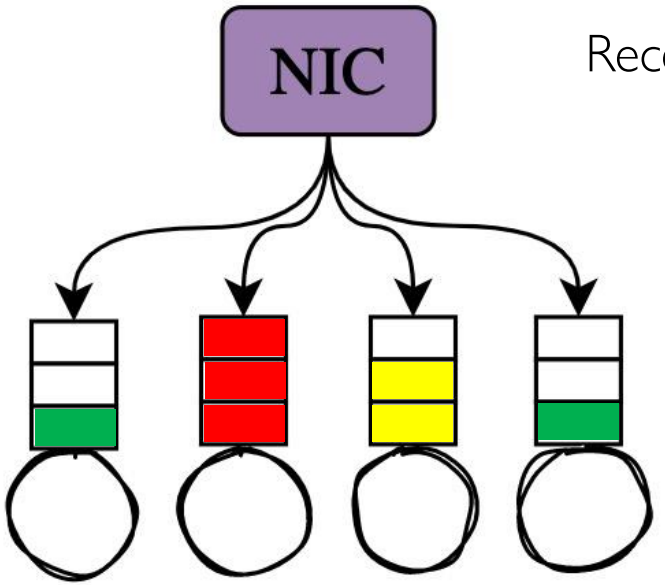
Load Imbalance

Kernel-native
d-FCFS runtime
Shared mem.

IX
[Belay et al., OSDI'14]

Bottlenecks in Prior RPC Schedulers

- Decentralized First Come First Serve (D-FCFS)



Receive Side Scaling (RSS)

Queues

Cores

Load Imbalance

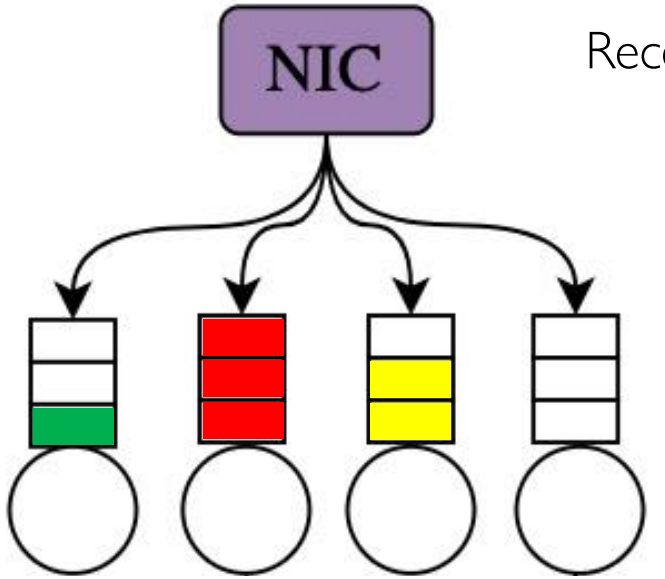
Kernel-native
d-FCFS runtime
Shared mem.

IX
[Belay et al., OSDI'14]

- SLO: Not Met
- Throughput: Low
- Core Utilization: Bad

Bottlenecks in Prior RPC Schedulers

- D-FCFS + work stealing



Receive Side Scaling (RSS)

Queues

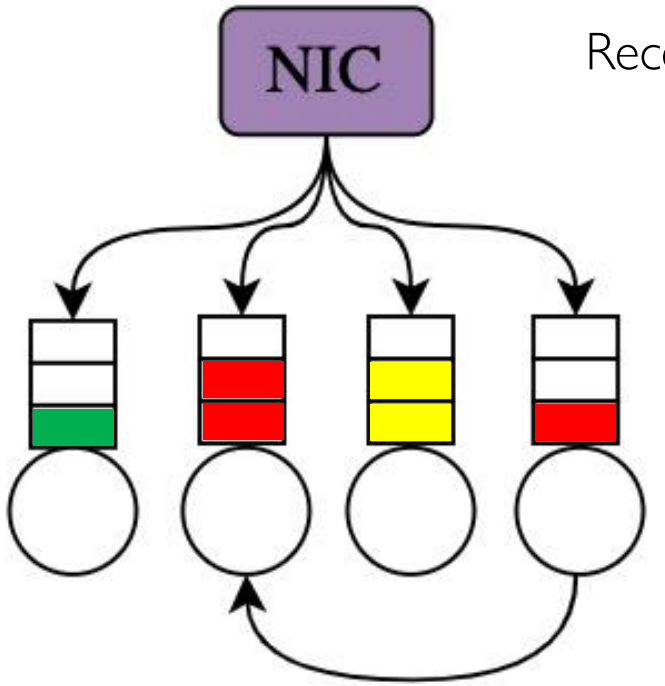
Cores

Work stealing
d-FCFS runtime
Shared mem.

ZygOS
[Prekas et al., SOSP'17]

Bottlenecks in Prior RPC Schedulers

- D-FCFS + work stealing

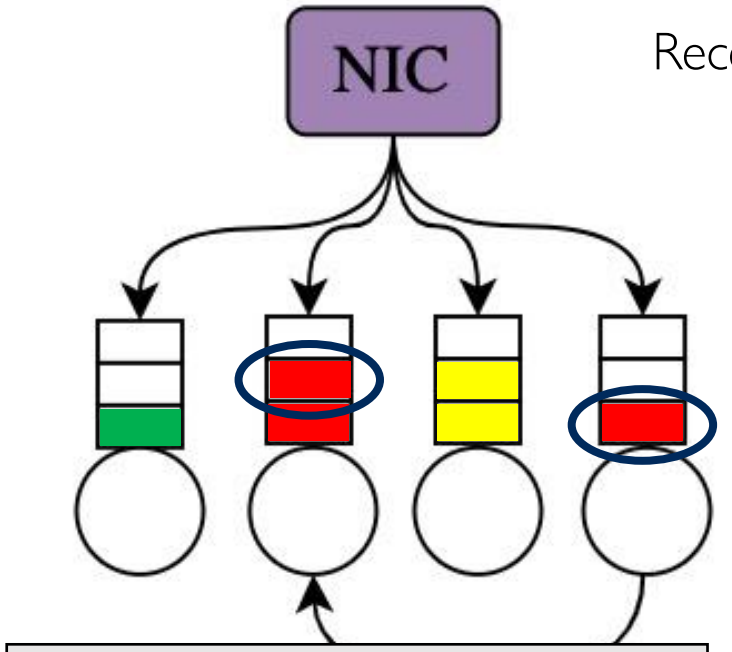


Work stealing
d-FCFS runtime
Shared mem.

ZygOS
[Prekas et al., SOSP'17]

Bottlenecks in Prior RPC Schedulers

- D-FCFS + work stealing



Receive Side Scaling (RSS)

Queues

Cores

Reactive – too late

SLO unaware

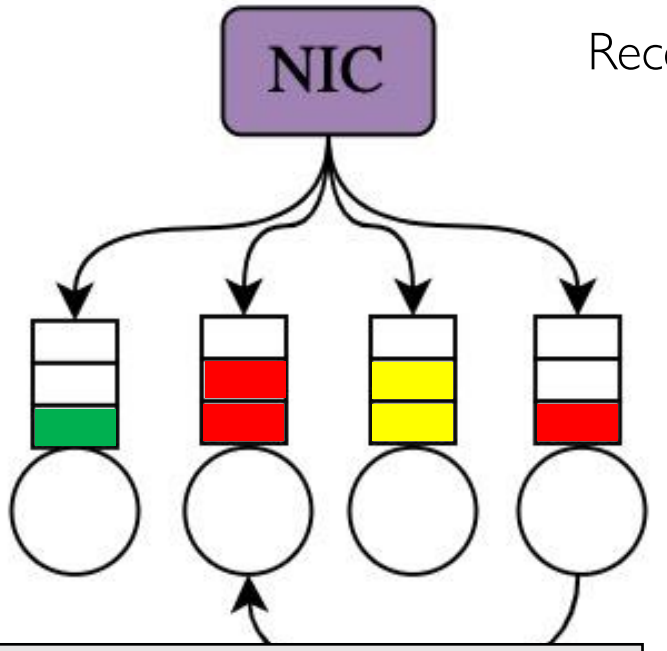
Work stealing
d-FCFS runtime
Shared mem.

ZygOS
[Prekas et al., SOSP'17]

- Core Utilization: Good
- SLO: Not Met

Bottlenecks in Prior RPC Schedulers

- D-FCFS + work stealing



Receive Side Scaling (RSS)

Queues

Cores

Work stealing
d-FCFS runtime
Shared mem.

ZygOS
[Prekas et al., SOSP'17]

- Core Utilization: Good
- SLO: Not Met
- Throughput: Medium

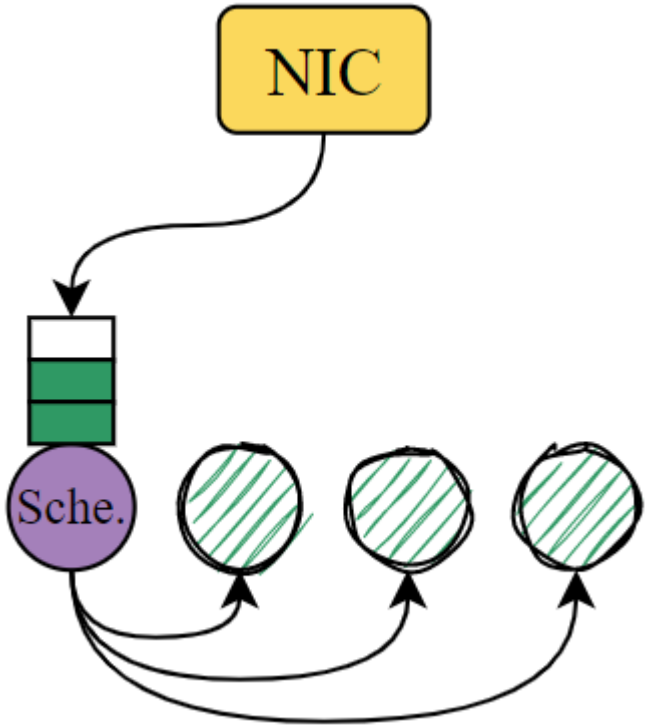
Reactive – too late

SLO unaware

Slow steal

Bottlenecks in Prior RPC Schedulers

- Centralized First Come First Serve (C-FCFS)

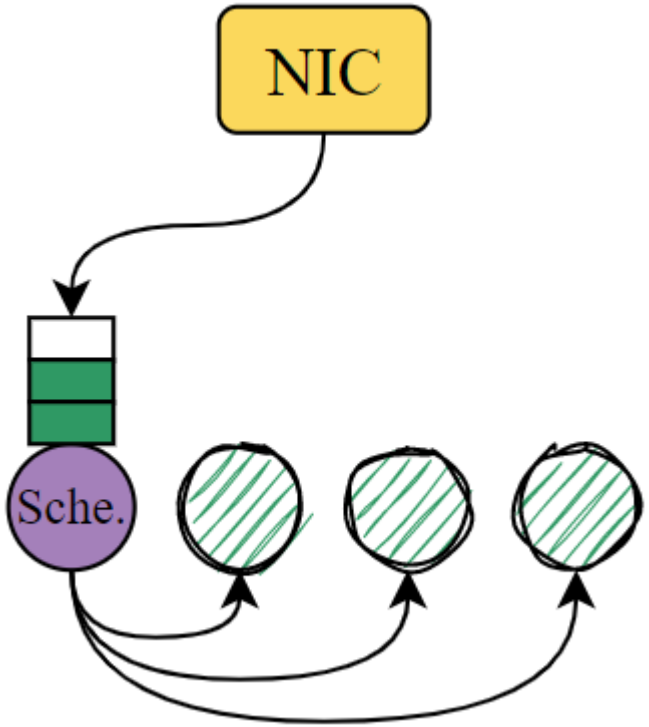


Preemption
c-FCFS runtime
Shared mem.

Shinjuku
[Kaffes et al., NSDI'19]

Bottlenecks in Prior RPC Schedulers

- Centralized First Come First Serve (C-FCFS)



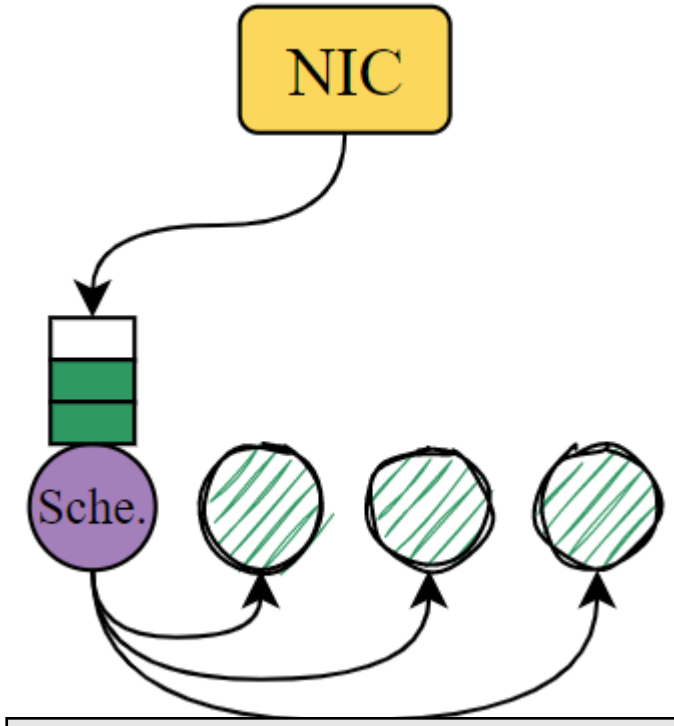
Preemption
c-FCFS runtime
Shared mem.

Shinjuku
[Kaffes et al., NSDI'19]

- SLO: Met
- Throughput: High
- Core Utilization: Good

Bottlenecks in Prior RPC Schedulers

- Centralized First Come First Serve (C-FCFS)



Scheduling core - Bottleneck

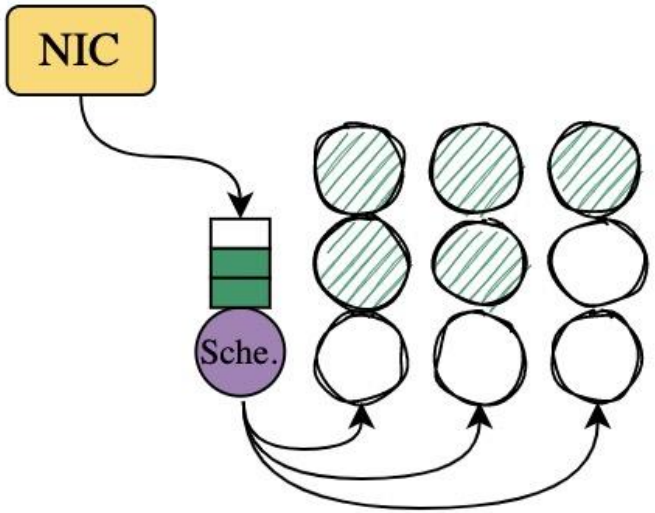
Preemption
c-FCFS runtime
Shared mem.

Shinjuku
[Kaffes et al., NSDI'19]

- SLO: Met
- Throughput: High
- Core Utilization: Good

Bottlenecks in Prior RPC Schedulers

- Centralized First Come First Serve (C-FCFS)



Scheduling core - Bottleneck

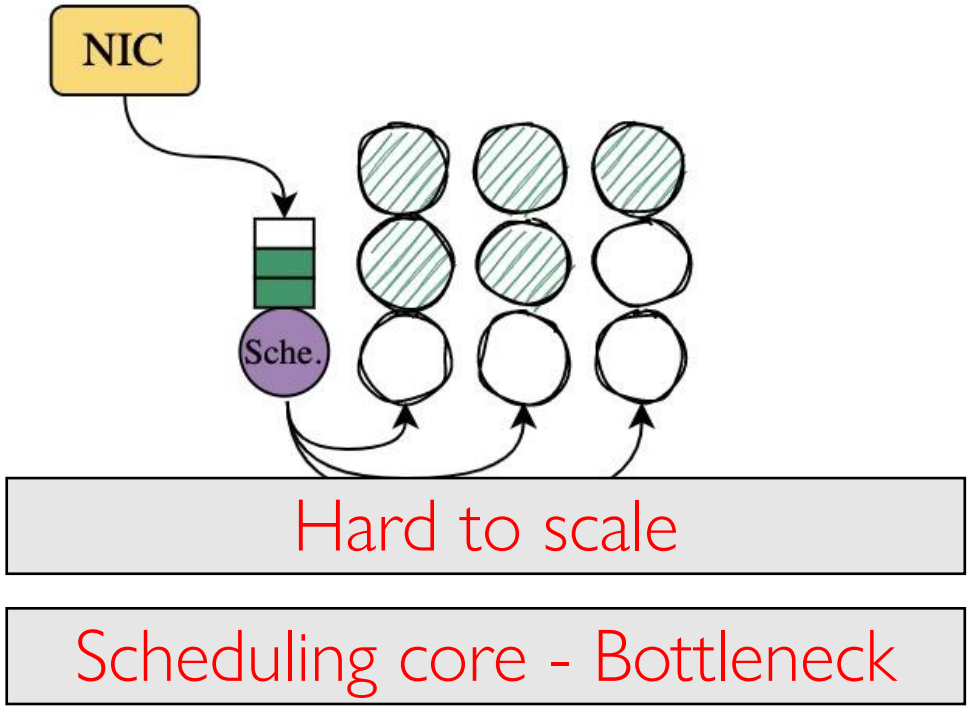
Preemption
c-FCFS runtime
Shared mem.

Shinjuku
[Kaffes et al., NSDI'19]

- SLO: Not Met
- Throughput: Low
- Core Utilization: Bad

Bottlenecks in Prior RPC Schedulers

- Centralized First Come First Serve (C-FCFS)



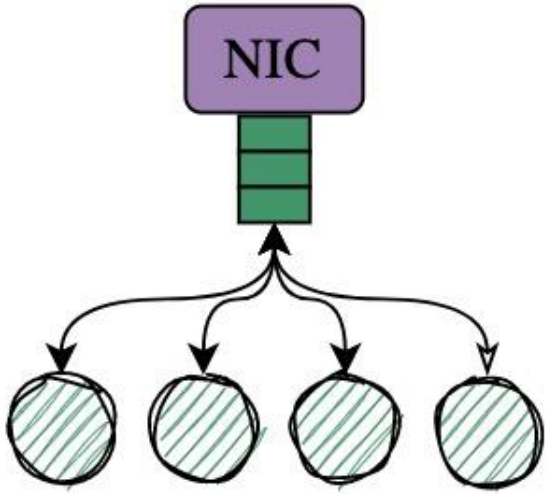
Preemption
c-FCFS runtime
Shared mem.

Shinjuku
[Kaffes et al., NSDI'19]

- SLO: Not Met
- Throughput: Low
- Core Utilization: Bad

Bottlenecks in Prior RPC Schedulers

- Centralized First Come First Serve (C-FCFS)



JBSQ algorithm
c-FCFS runtime
Cache coherence

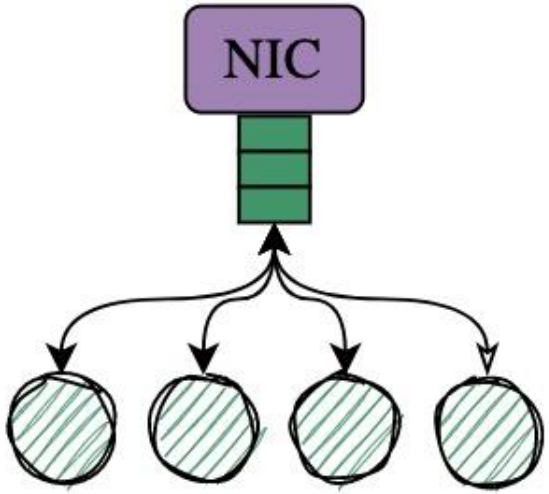
Nebula
[Sutherland et al., ISCA'20]

JBSQ algorithm
c-FCFS runtime
Direct register

nanoPU
[Ibanez et al., OSDI'21]

Bottlenecks in Prior RPC Schedulers

- Centralized First Come First Serve (C-FCFS)



- SLO: Met
- Throughput: High
- Core Utilization: Good

JBSQ algorithm
c-FCFS runtime
Cache coherence

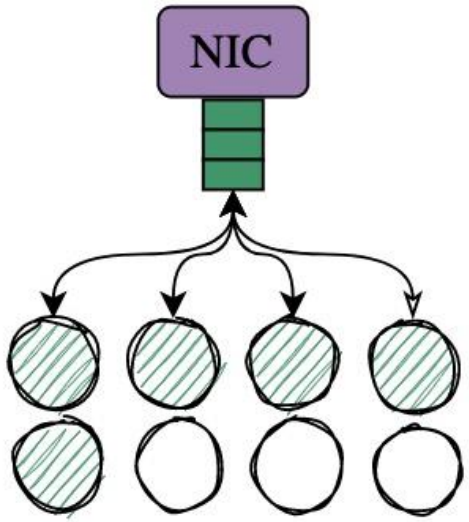
Nebula
[Sutherland et al., ISCA'20]

JBSQ algorithm
c-FCFS runtime
Direct register

nanoPU
[Ibanez et al., OSDI'21]

Bottlenecks in Prior RPC Schedulers

- Centralized First Come First Serve (C-FCFS)



Coherence inscalability

Fixed scheduling algorithm

JBSQ algorithm
c-FCFS runtime
Cache coherence

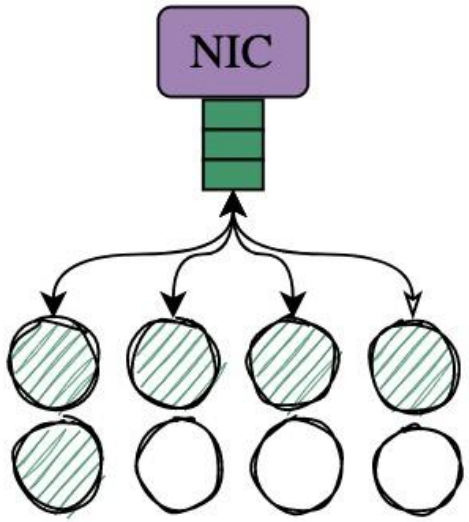
Nebula
[Sutherland et al., ISCA'20]

JBSQ algorithm
c-FCFS runtime
Direct register

nanoPU
[Ibanez et al., OSDI'21]

Bottlenecks in Prior RPC Schedulers

- Centralized First Come First Serve (C-FCFS)



- SLO: Not Met
- Throughput: Low
- Core Utilization: Bad

Coherence inscalability

Fixed scheduling algorithm

JBSQ algorithm
c-FCFS runtime
Cache coherence

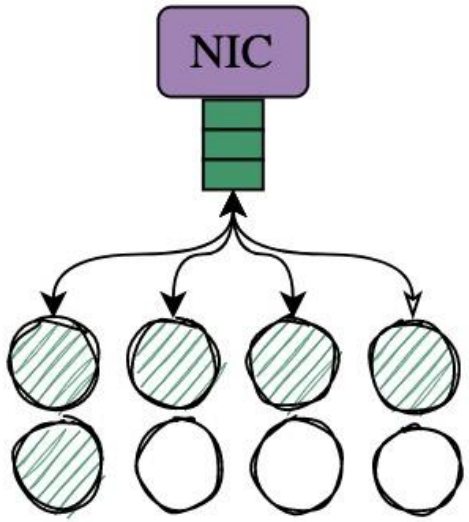
Nebula
[Sutherland et al., ISCA'20]

JBSQ algorithm
c-FCFS runtime
Direct register

nanoPU
[Ibanez et al., OSDI'21]

Bottlenecks in Prior RPC Schedulers

- Centralized First Come First Serve (C-FCFS)



- SLO: Not Met
- Throughput: Low
- Core Utilization: Bad

Coherence inscalability

Fixed scheduling algorithm

Hard to scale

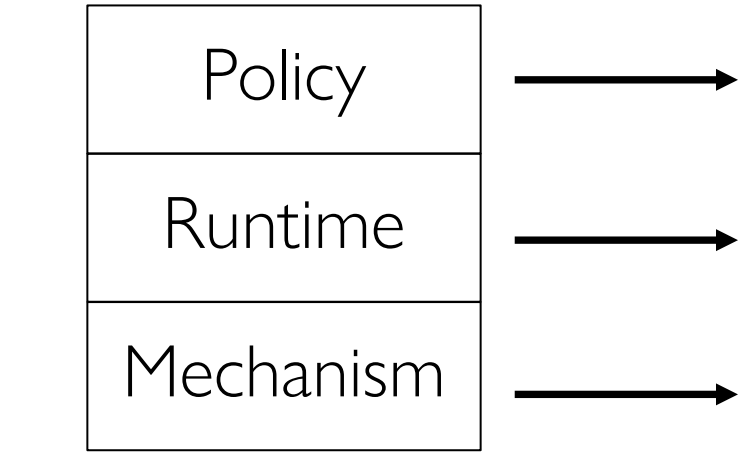
JBSQ algorithm
c-FCFS runtime
Cache coherence

Nebula
[Sutherland et al., ISCA'20]

JBSQ algorithm
c-FCFS runtime
Direct register

nanoPU
[Ibanez et al., OSDI'21]

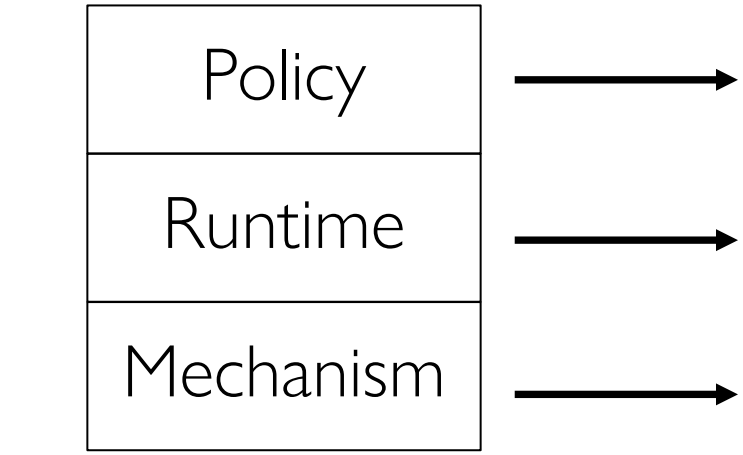
Design Challenges: Summary



Scheduling Sub-system Abstraction

Challenge 1: SLO-unaware, reactive or not adaptive

Design Challenges: Summary

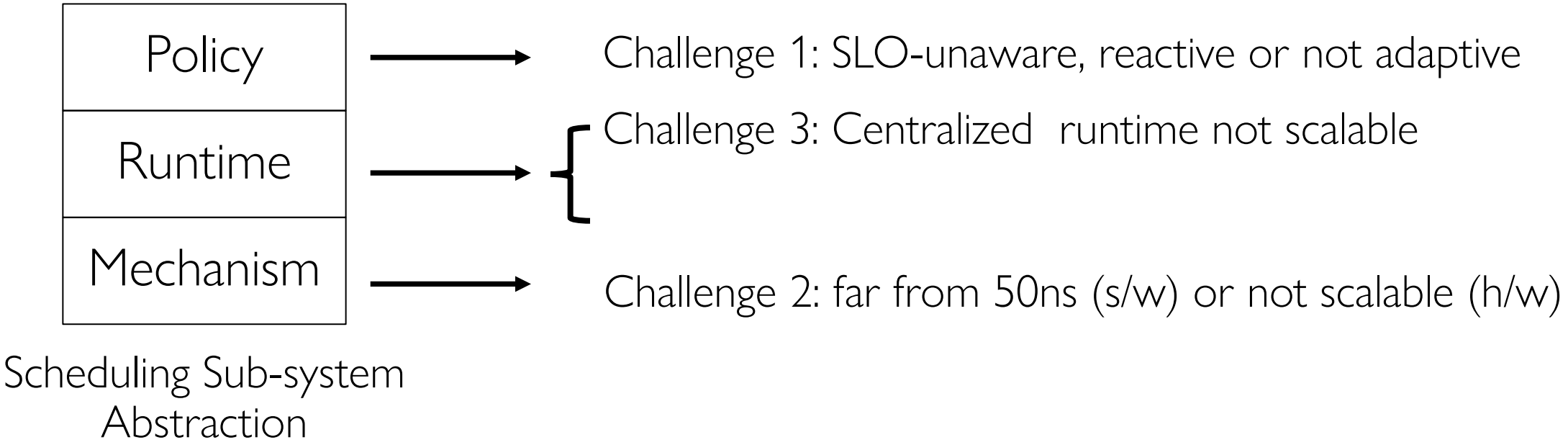


Scheduling Sub-system Abstraction

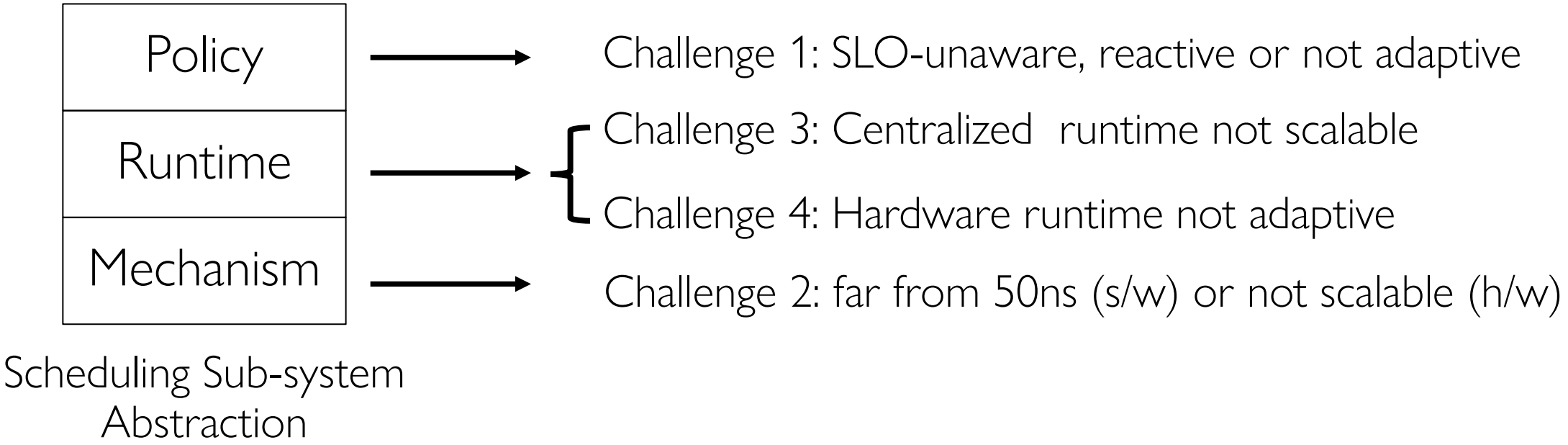
Challenge 1: SLO-unaware, reactive or not adaptive

Challenge 2: far from 50ns (s/w) or not scalable (h/w)

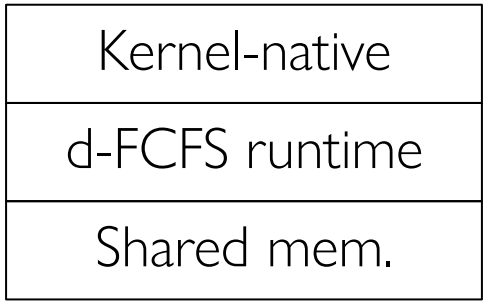
Design Challenges: Summary



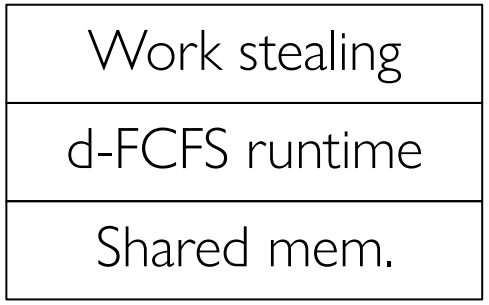
Design Challenges: Summary



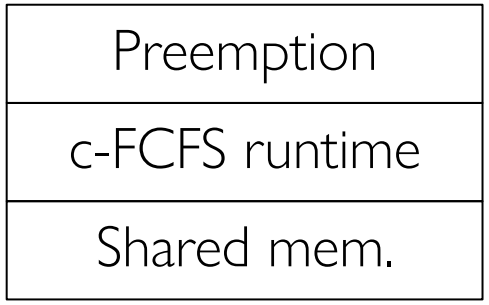
Altocumulus: High-Level Perspective



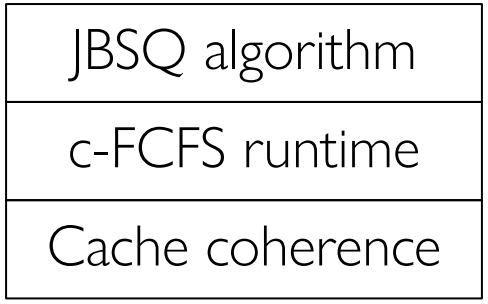
IX
[Belay et al., OSDI'14]



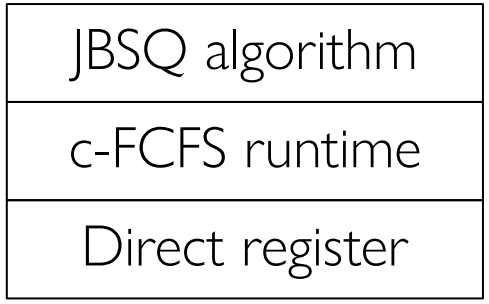
ZygOS
[Prekas et al., SOSP'17]



Shinjuku
[Kaffes et al., NSDI'19]

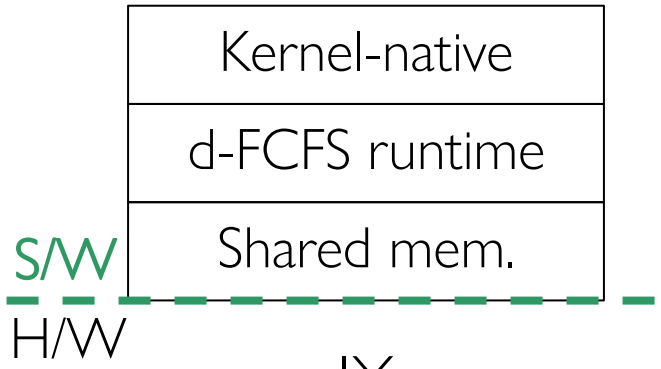


Nebula
[Sutherland et al., ISCA'20]

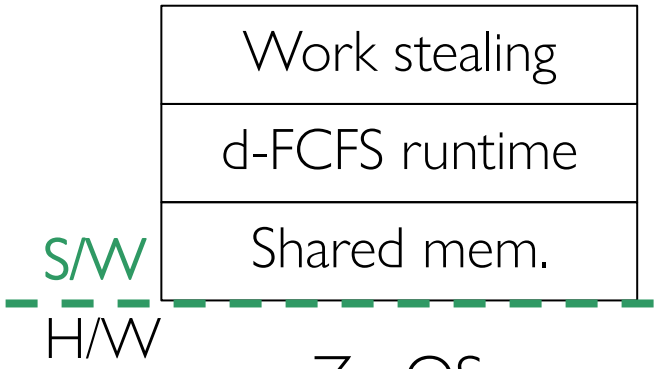


nanoPU
[Ibanez et al., OSDI'21]

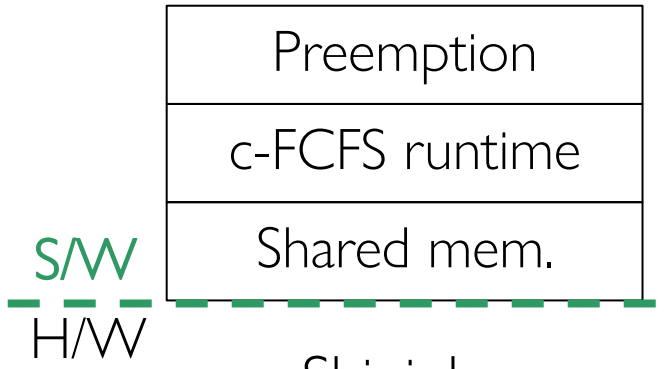
Altocumulus: High-Level Perspective



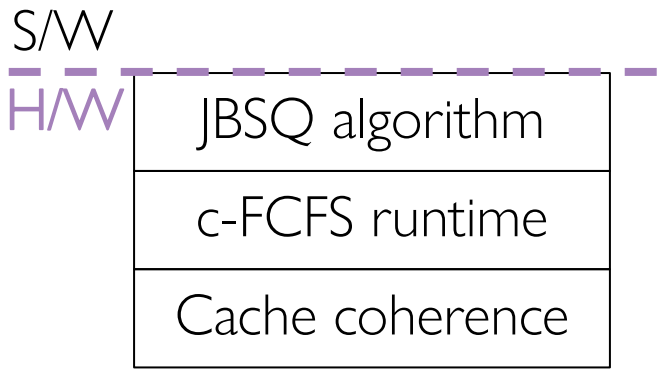
IX
[Belay et al., OSDI'14]



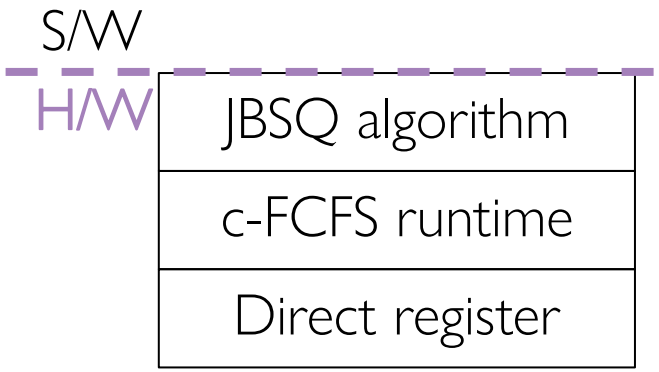
ZygOS
[Prekas et al., SOSP'17]



Shinjuku
[Kaffes et al., NSDI'19]

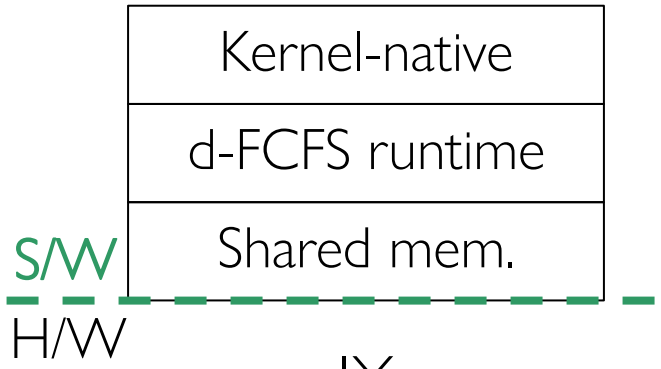


Nebula
[Sutherland et al., ISCA'20]

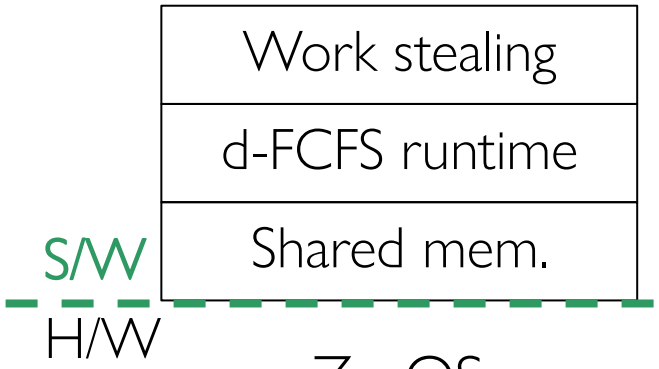


nanoPU
[Ibanez et al., OSDI'21]

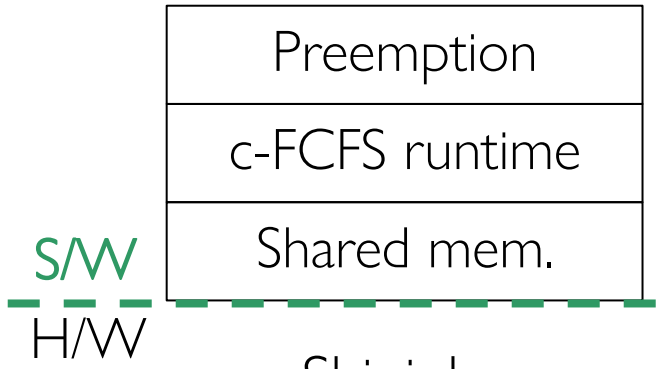
Altocumulus: High-Level Perspective



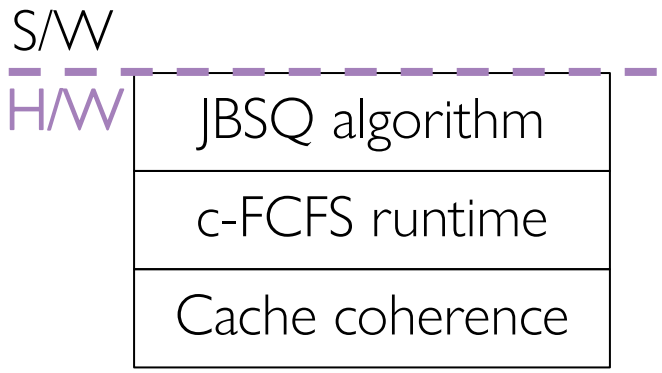
IX
[Belay et al., OSDI'14]



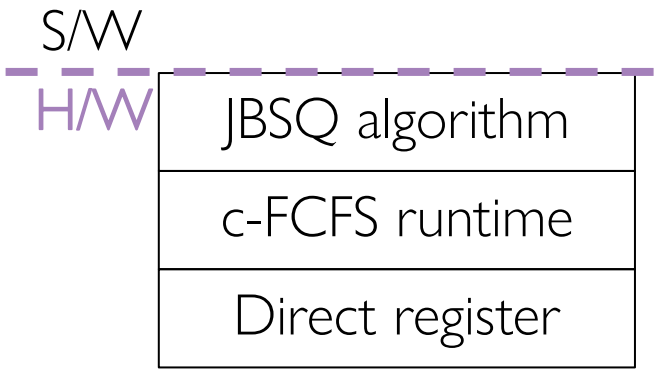
ZygOS
[Prekas et al., SOSP'17]



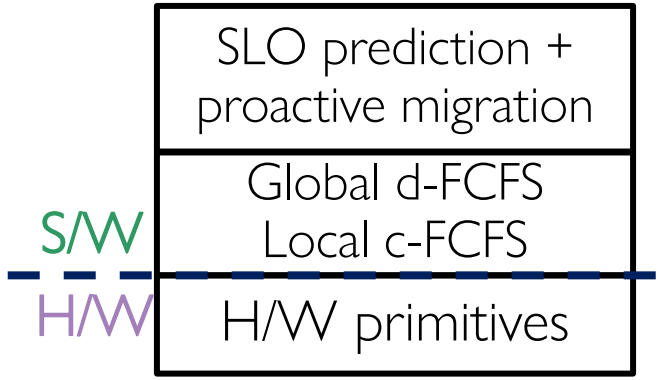
Shinjuku
[Kaffes et al., NSDI'19]



Nebula
[Sutherland et al., ISCA'20]

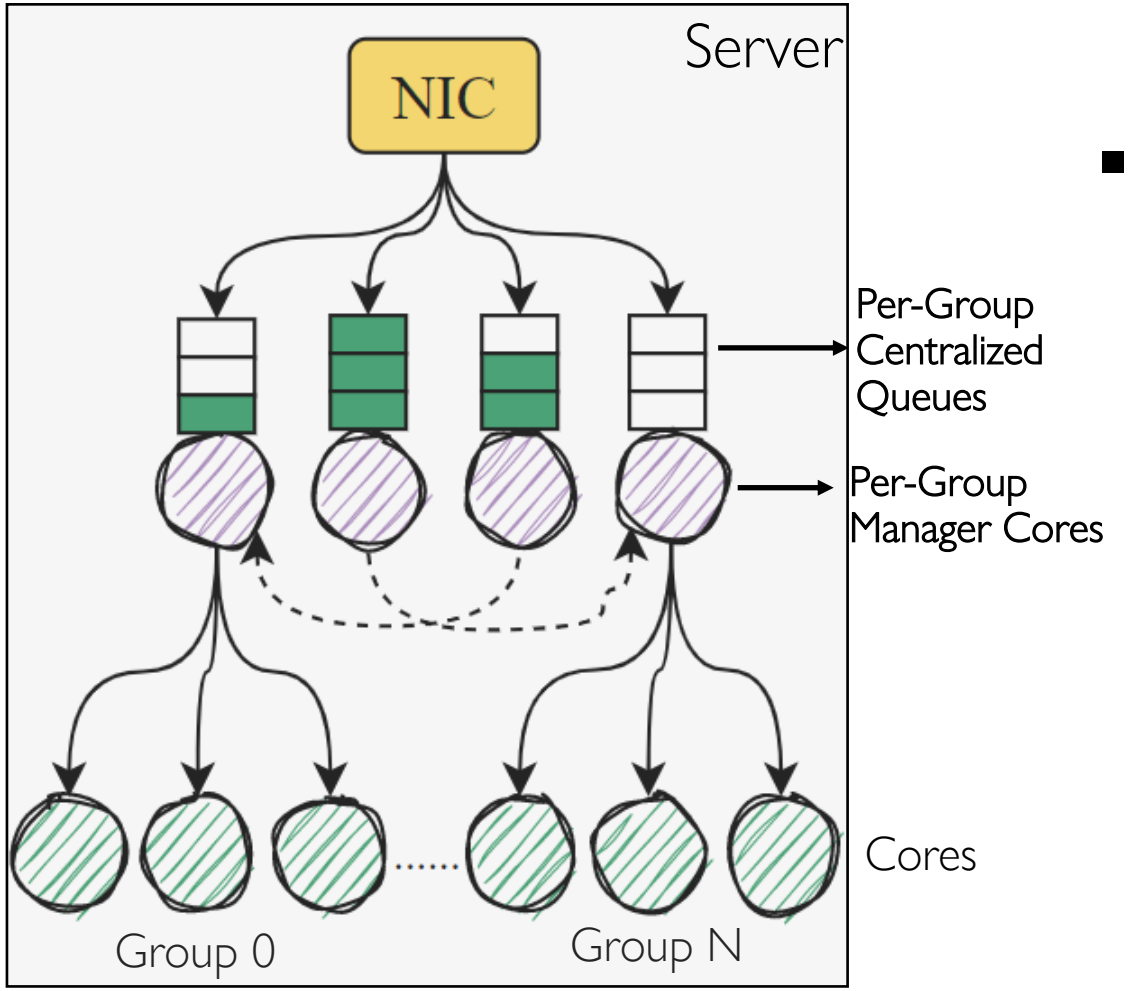


nanoPU
[Ibanez et al., OSDI'21]



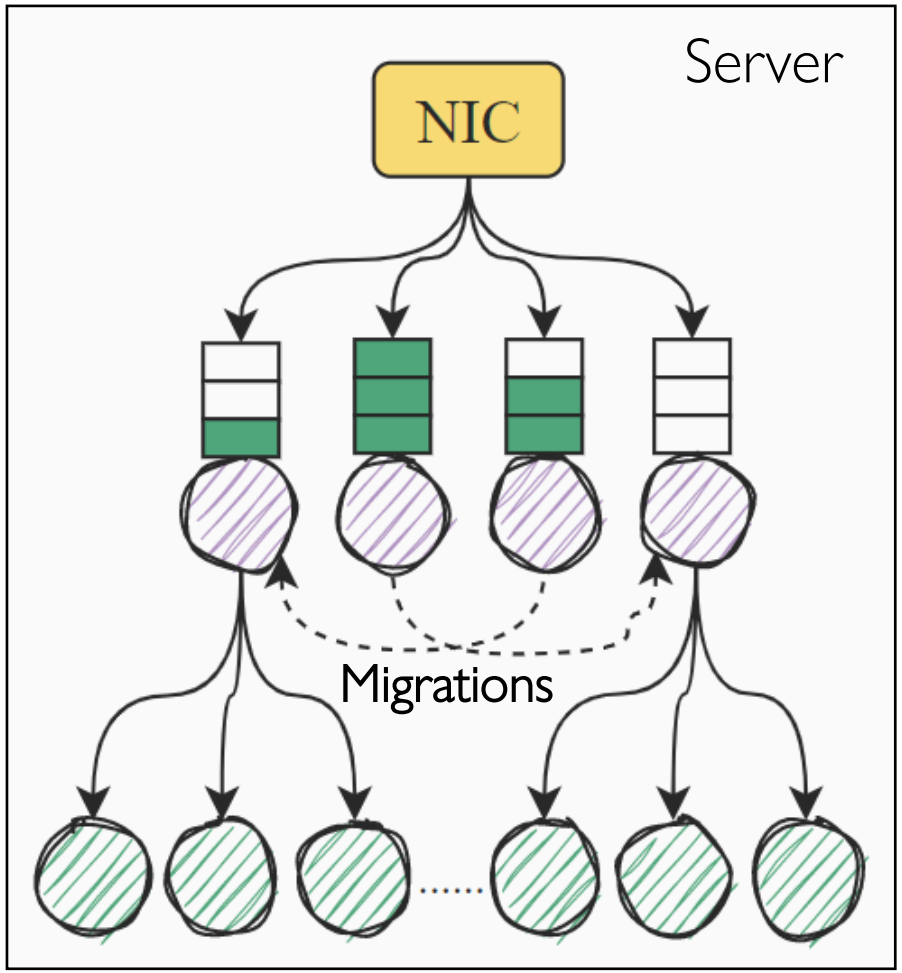
Altocumulus

Altocumulus Architecture



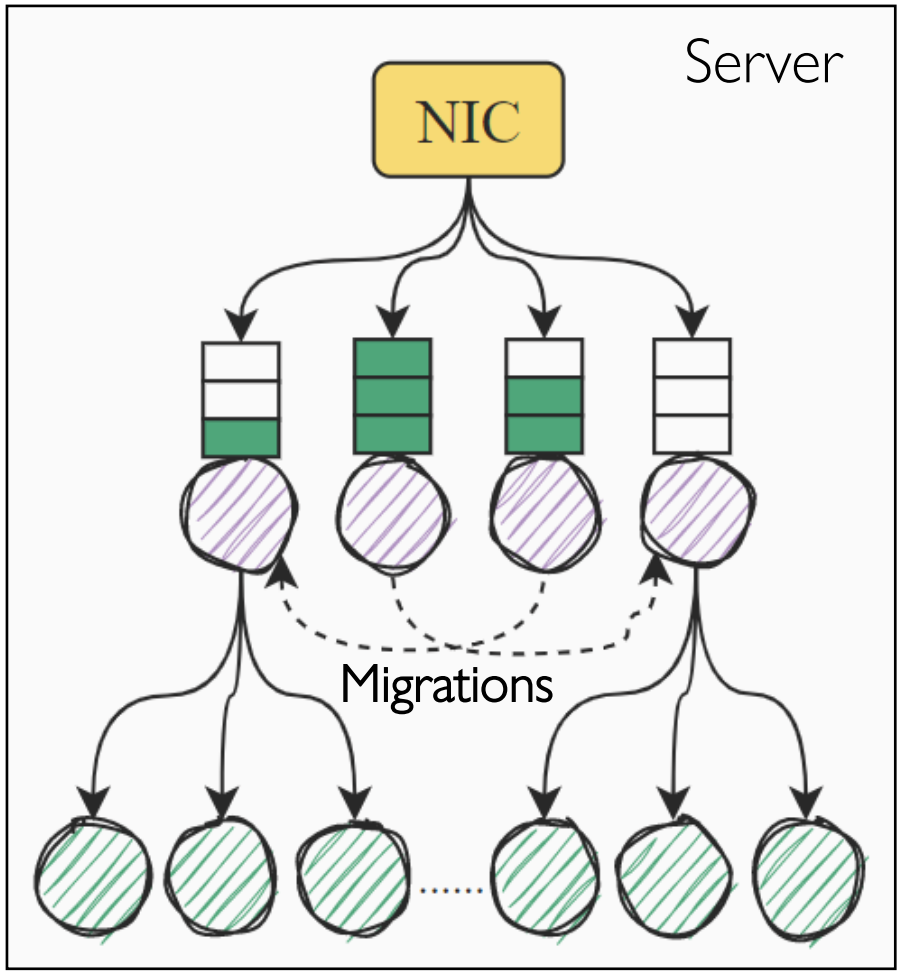
- Multi-tiered global D-FCFS local C-FCFS
  - Each group: 1 queue + 1 manager (purple) + several workers (green)

Altocumulus Architecture



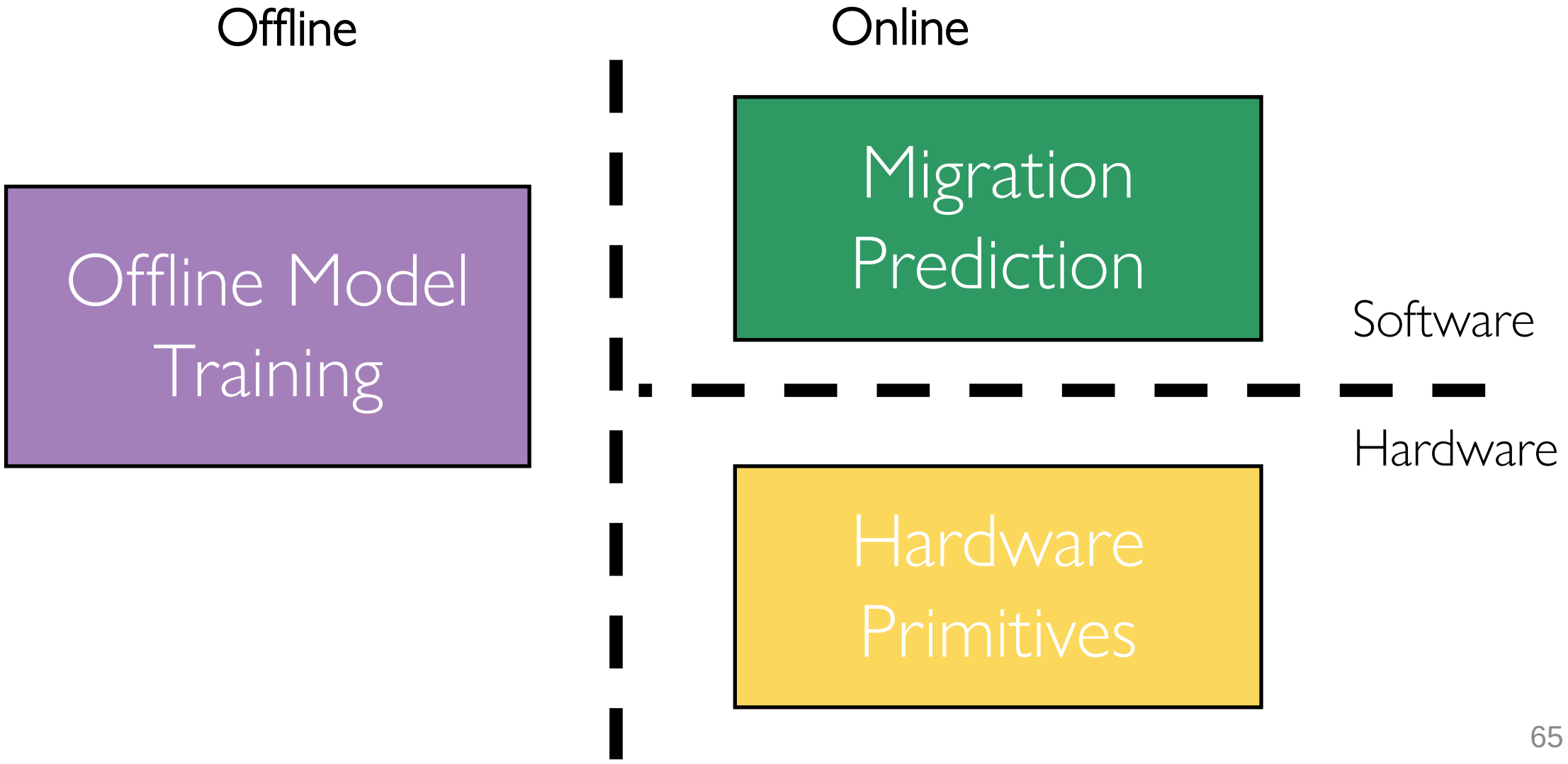
- Multi-tiered global D-FCFS local C-FCFS
  - Each group: 1 queue + 1 manager (purple) + several workers (green)
- Proactive & hardware-assisted work stealing across manager cores

Altocumulus Architecture

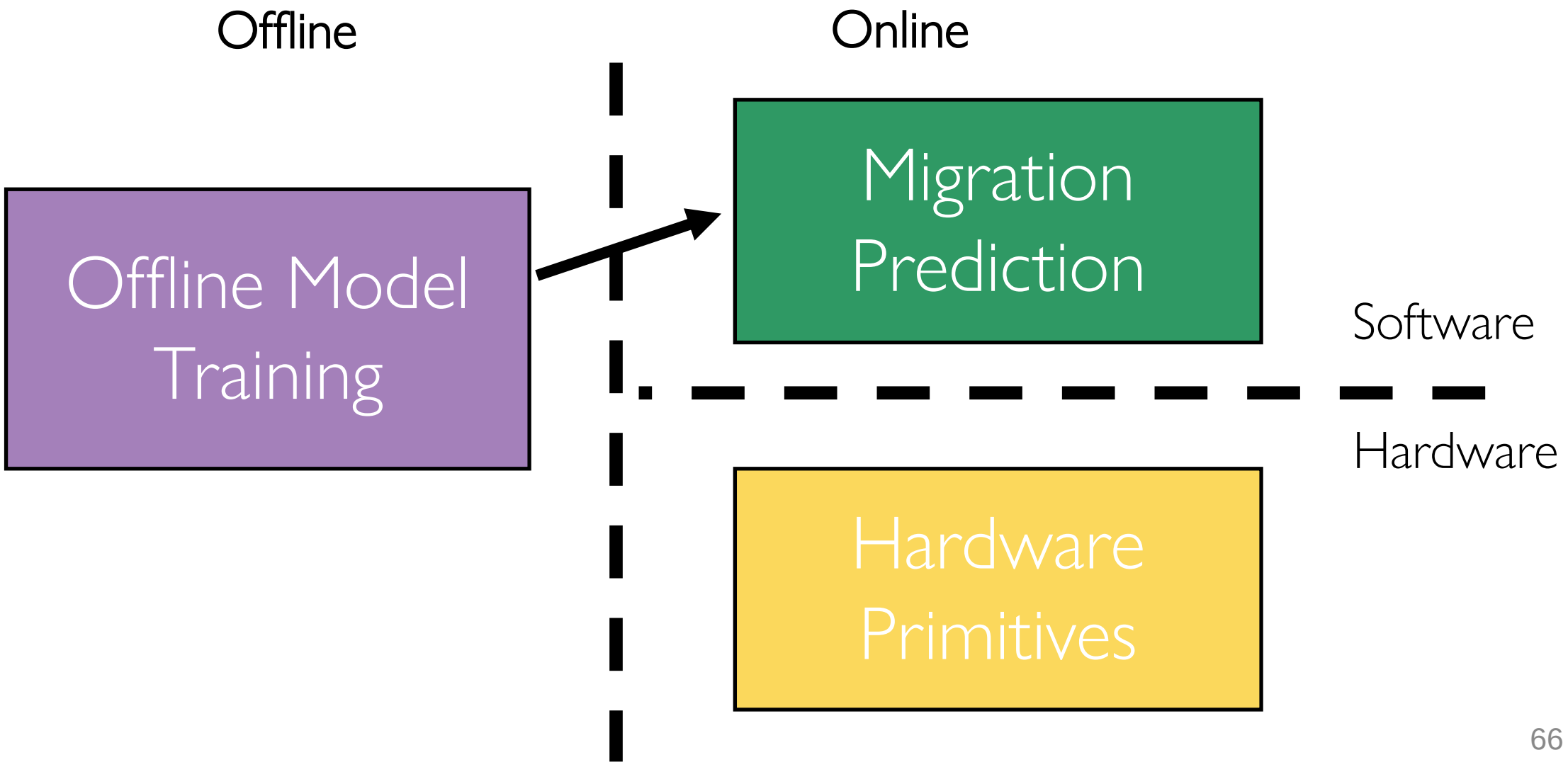


- Multi-tiered global D-FCFS local C-FCFS
  - Each group: 1 queue + 1 manager (purple) + several workers (green)
- Proactive & hardware-assisted work stealing across manager cores
- Compatible with commodity multi-queue NIC with RSS support

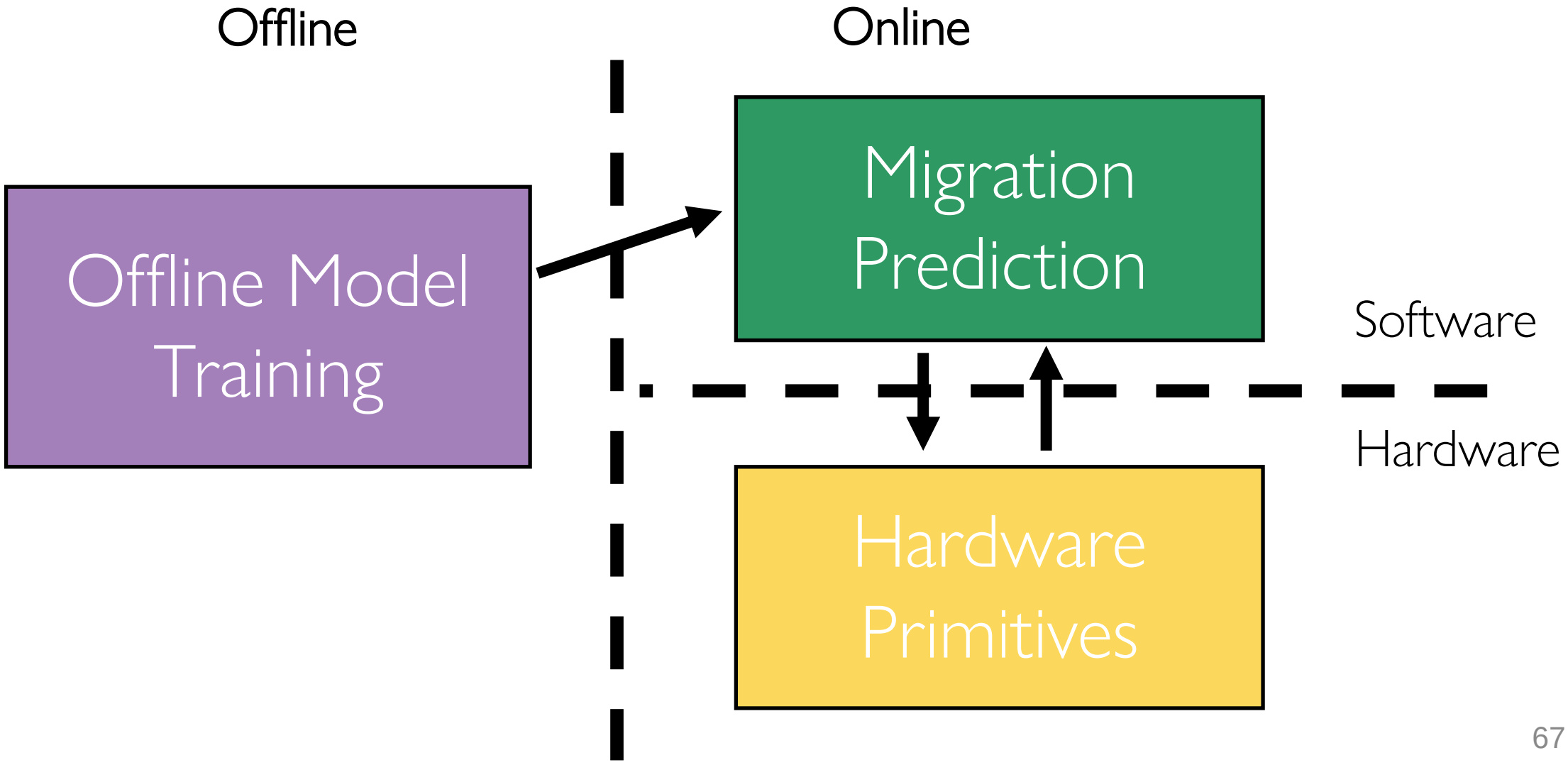
Altocumulus System



Altocumulus System



Altocumulus System

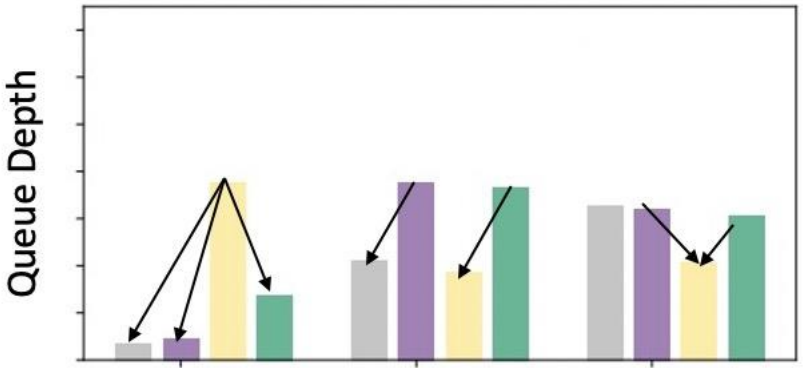


Migration Prediction

1. Periodically synchronize system states across manager cores
2. Pick appropriate threshold trained offline
3. Poll queues to check if threshold is met
4. Decide how many RPCs to migrate to which queue(s)

Migration Prediction

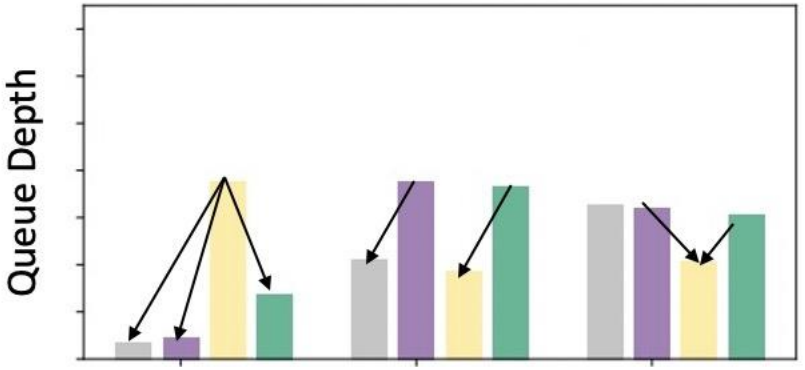
1. Periodically synchronize system states
2. Pick appropriate threshold trained offline
3. Poll queues to check if threshold is met
4. Decide how many RPCs to migrate to which queue(s)



Heuristics: multi-queue load pattern classification

Migration Prediction

1. Periodically synchronize system states
2. Pick appropriate threshold trained offline
3. Poll queues to check if threshold is met



Heuristics: multi-queue load pattern classification

4. Decide how many RPCs to migrate to which queue(s)
5. Pass decisions to h/w primitives and trigger migration h/w messages
6. Repeat Step 1 (Period as short as 50ns due to messaging offloaded to h/w)

Migration Prediction

1. Periodically synchronize system states
2. Pick appropriate threshold trained offline
3. Poll queues to check if threshold is met
4. Decide how many RPCs to migrate to which queue(s)
5. Pass decisions to h/w primitives and trigger migration h/w messages
6. Repeat Step 1 (Period as short as 50ns due to messaging offloaded to h/w)

How we train SLO violation prediction model?

Heuristics: multi-queue load pattern classification

How we offload messaging to h/w ?

Hardware Primitives: Direct Register Messaging

- Register-to-register migration messaging, bypassing cache coherence protocol
  - Inspired by [Sanchez et al., ASPLOS'10] & [Ibanez et al., OSDI'21] w/ several opt.

Hardware Primitives: Direct Register Messaging

- Register-to-register migration messaging, bypassing cache coherence protocol
  - Inspired by [Sanchez et al., ASPLOS'10] & [Ibanez et al., OSDI'21] w/ several opt.
- Remain RPC message payload in LLC/memory -- Only move RPC descriptor
  - Reduce latency overhead and traffic per migration (up to 140x)

Hardware Primitives: Direct Register Messaging

- Register-to-register migration messaging, bypassing cache coherence protocol
  - Inspired by [Sanchez et al., ASPLOS'10] & [Ibanez et al., OSDI'21] w/ several opt.
- Remain RPC message payload in LLC/memory -- Only move RPC descriptor
  - Reduce latency overhead and traffic per migration (up to 140x)
- Each request being migrated at most once
  - Avoid livelock and unnecessary scheduling traffic

Hardware Primitives: Direct Register Messaging

- Register-to-register migration messaging, bypassing cache coherence protocol
  - Inspired by [Sanchez et al., ASPLOS'10] & [Ibanez et al., OSDI'21] w/ several opt.
- Remain RPC message payload in LLC/memory -- Only move RPC descriptor
  - Reduce latency overhead and traffic per migration (up to 140x)
- Each request being migrated at most once
  - Avoid livelock and unnecessary scheduling traffic
- Batch multiple descriptors per message
  - Improve hardware efficiency

Migration Prediction

1. Periodically synchronize system states
2. Pick appropriate threshold trained offline
3. Poll queues to check if threshold is met
4. Decide how many RPCs to migrate to which queue(s)
5. Pass decisions to h/w primitives and trigger migration h/w messages
6. Repeat Step 1 (Period as short as 50ns due to messaging offloaded to h/w)

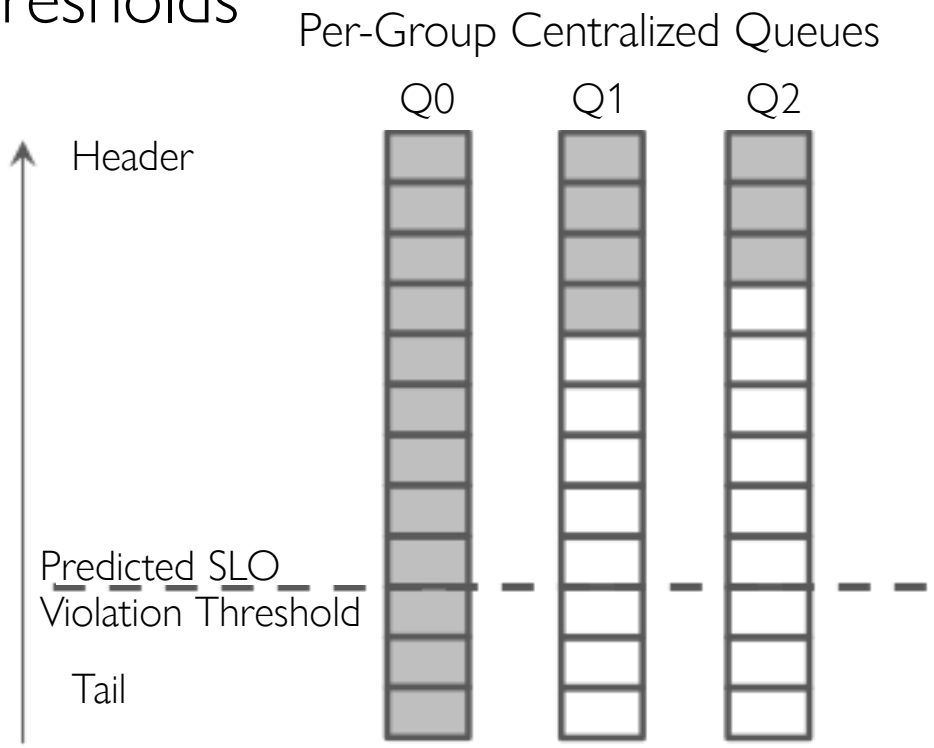
How we train SLO violation prediction model?

Heuristics: multi-queue load pattern classification

~~How we offload messaging to h/w ?~~

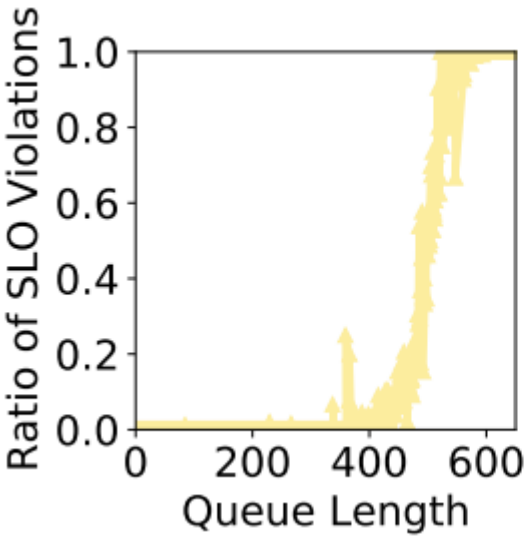
Offline Model Training

- Determine a vector of SLO violation thresholds
  - Use queuing theory assisted w/ simulation
- Different threshold per:
  - Load status
  - Service time distribution
  - Arrival pattern
  - Number of cores

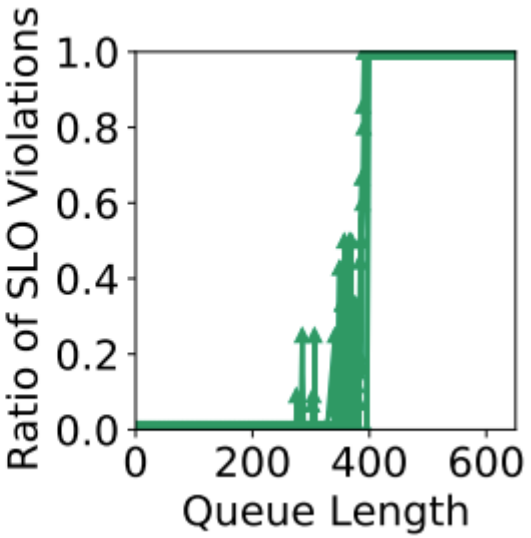


Offline Model Training

- Trade-off: Prediction accuracy V.S. migration effectiveness



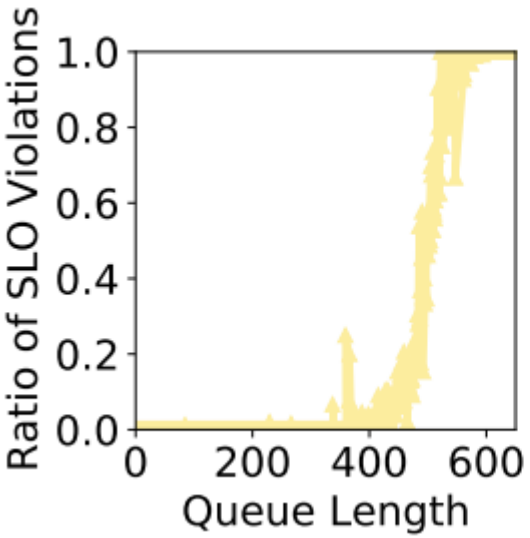
Uniform, Poisson, 64 cores



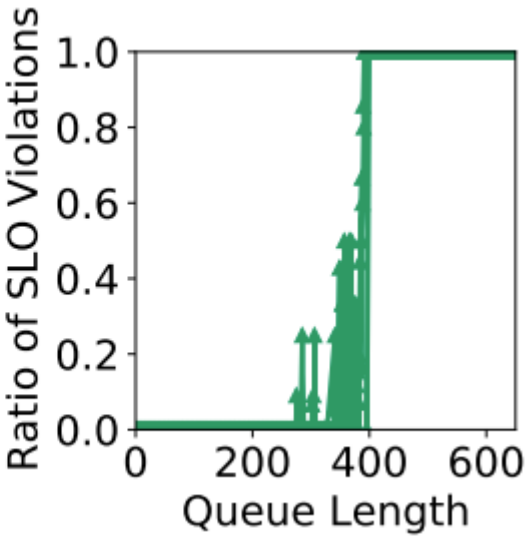
Bi-modal, Poisson, 64 cores

Offline Model Training

- Trade-off: Prediction accuracy V.S. migration effectiveness



Uniform, Poisson, 64 cores



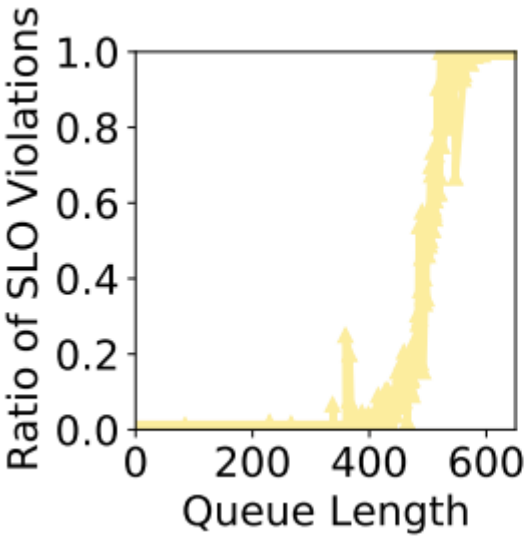
Bi-modal, Poisson, 64 cores

A typical SLO value:

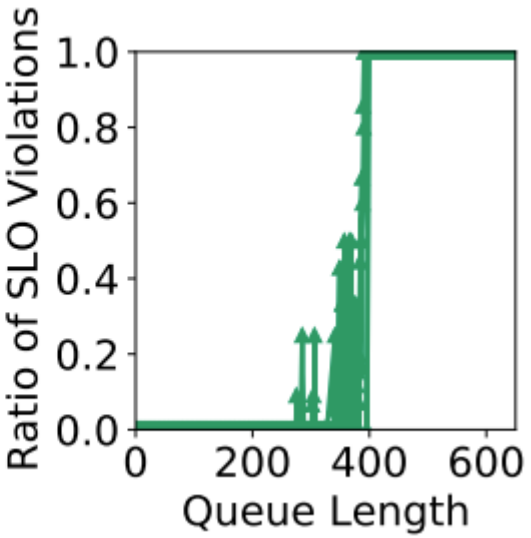
99th% latency <= 10 x average latency

Offline Model Training

- Trade-off: Prediction accuracy V.S. migration effectiveness



Uniform, Poisson, 64 cores



Bi-modal, Poisson, 64 cores

A typical SLO value:

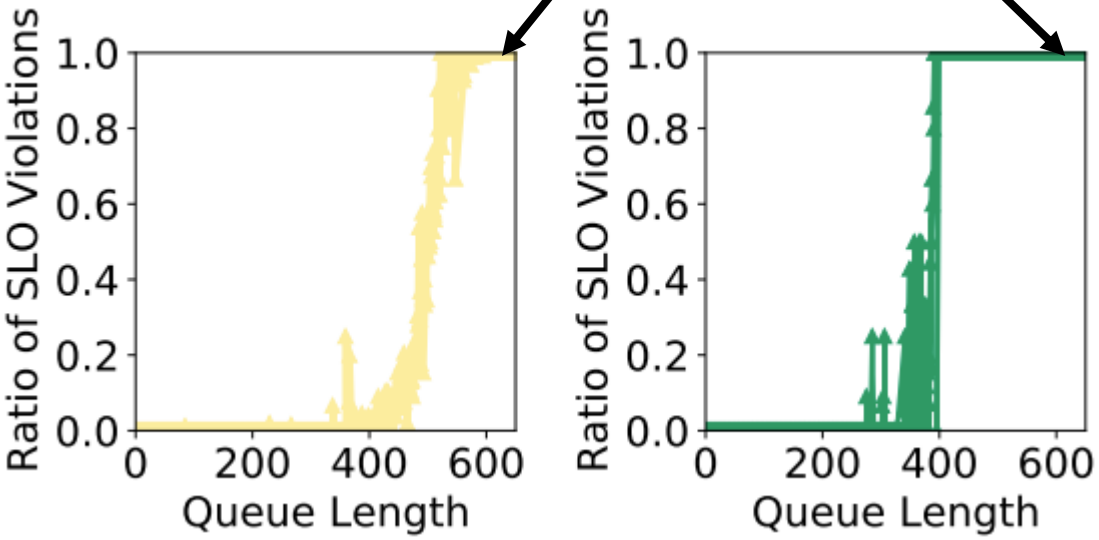
99th% latency <= 10 x average latency

Naive prediction approach:

Threshold = 10 x number of cores

Offline Model Training

- Trade-off: Prediction accuracy V.S. migration effectiveness
  - Prediction accuracy per migration: 100%
  - Effectiveness (Capture ? % of SLO violation): ~0%



Uniform, Poisson, 64 cores Bi-modal, Poisson, 64 cores

A typical SLO value:

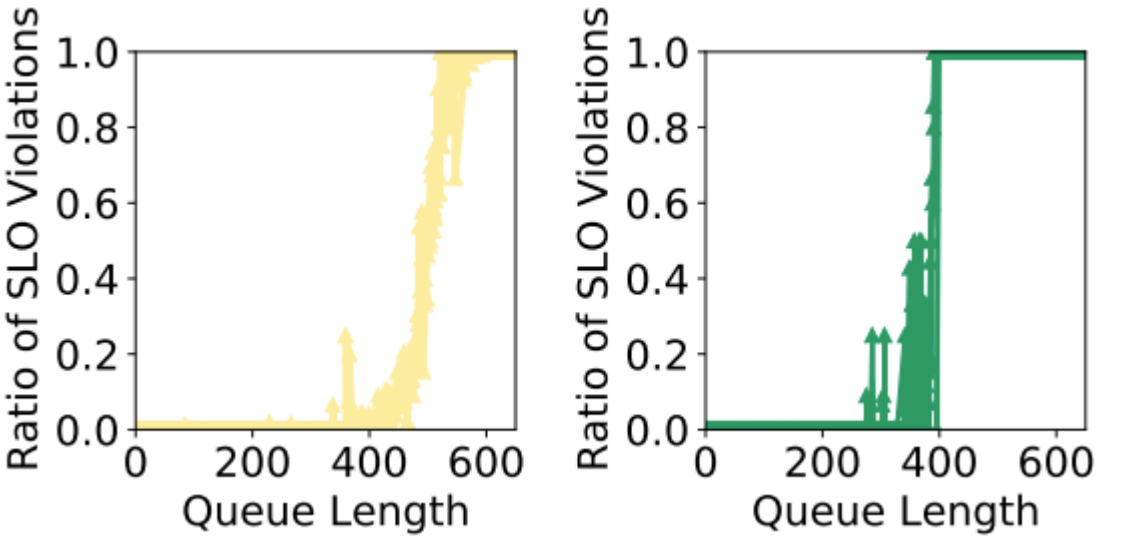
99th% latency <= 10 x average latency

Naive prediction approach:

Threshold = 10 x number of cores

Offline Model Training

- Trade-off: Prediction accuracy V.S. migration effectiveness



Uniform, Poisson, 64 cores

Bi-modal, Poisson, 64 cores

A typical SLO value:

99th% latency <= 10 x average latency

~~Naive prediction approach~~:

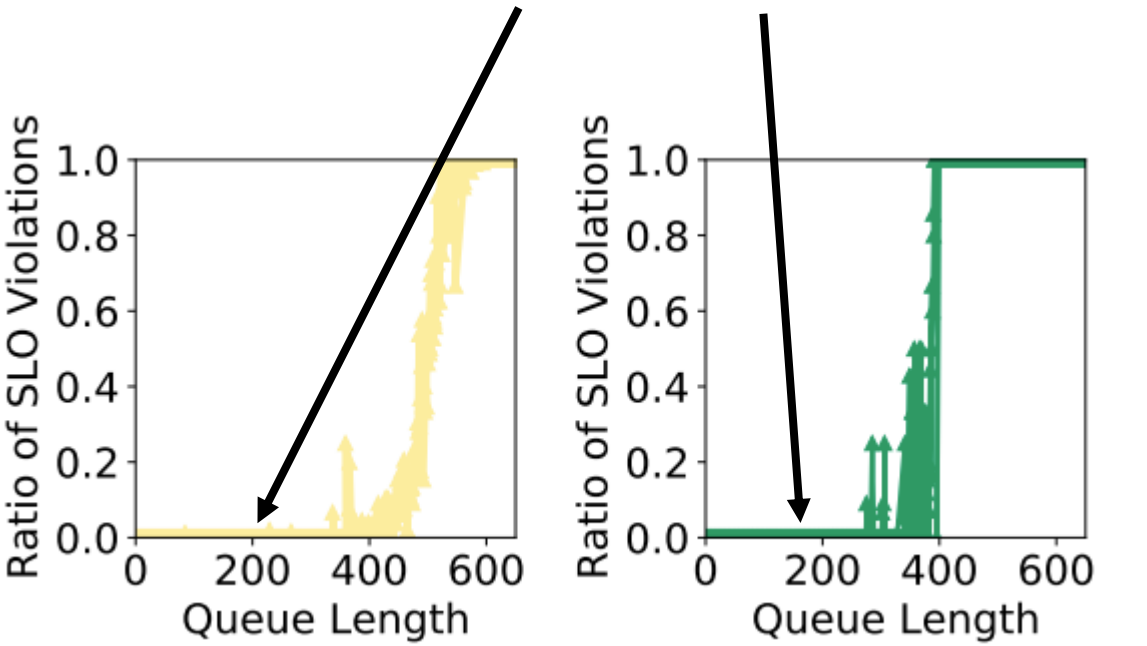
~~Threshold = 10 x number of cores~~

Aggressive prediction approach:

Threshold = first SLO-violated queue length

Offline Model Training

- Trade-off: Prediction accuracy V.S. migration effectiveness
  - Prediction accuracy per migration: ~0%
  - Effectiveness (Capture ? % of SLO violation): 100%



Uniform, Poisson, 64 cores

Bi-modal, Poisson, 64 cores

A typical SLO value:

99th% latency <= 10 x average latency

~~Naive prediction approach~~:

~~Threshold = 10 x number of cores~~

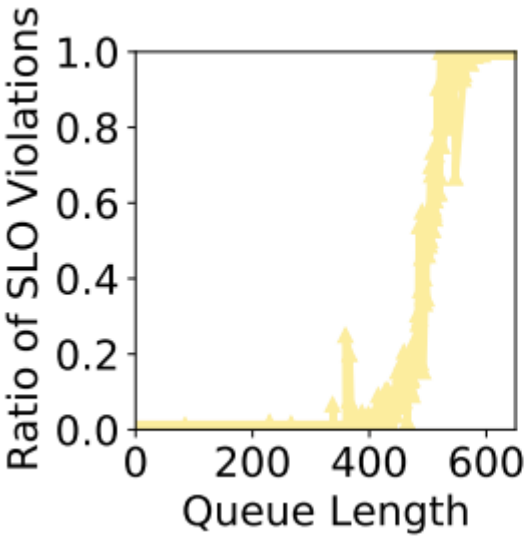
Aggressive prediction approach:

Threshold = first SLO-violated queue length

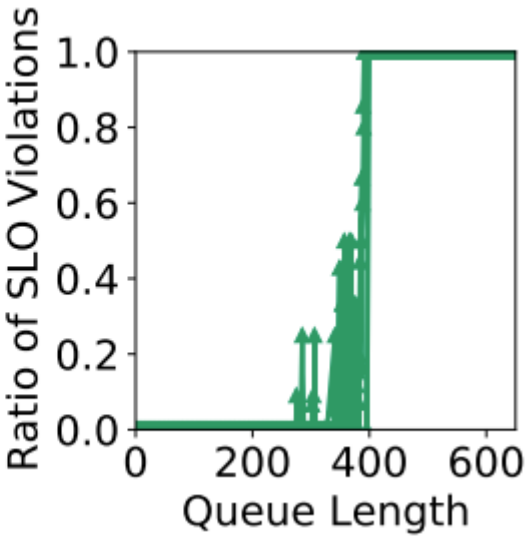
Offline Model Training

- Trade-off: Prediction accuracy V.S. migration effectiveness

We characterize thresholds for all system states offline & dynamically select threshold online



Uniform, Poisson, 64 cores



Bi-modal, Poisson, 64 cores

A typical SLO value:

99th% latency <= 10 x average latency

~~Naive prediction approach~~:

~~Threshold = 10 x number of cores~~

~~Aggressive prediction approach:~~

~~Threshold = first SLO-violated queue length~~

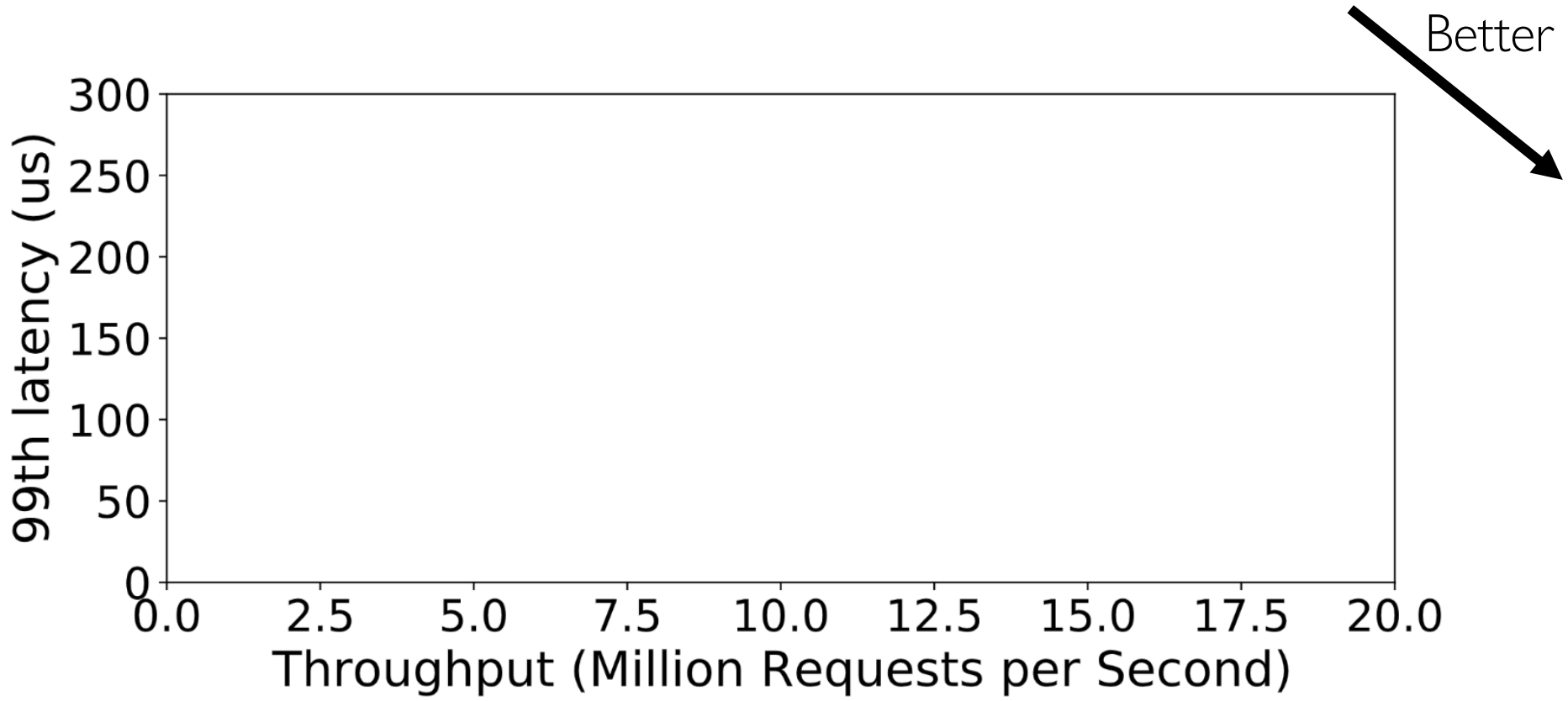
Methodology

- Baselines
  - D-FCFS + work stealing system
    - ZygOS [Prekas et al, SOSP'17]
  - C-FCFS S/W based system
    - Shinjuku [Kaffes et al, NSDI'19]
  - C-FCFS H/W based system
    - Nebula [Sutherland et al, ISCA'20]
    - nanoPU [Ibanez et al, OSDI'21]
- Altocumulus configuration (Simulation)
  - Commodity RSS NIC +

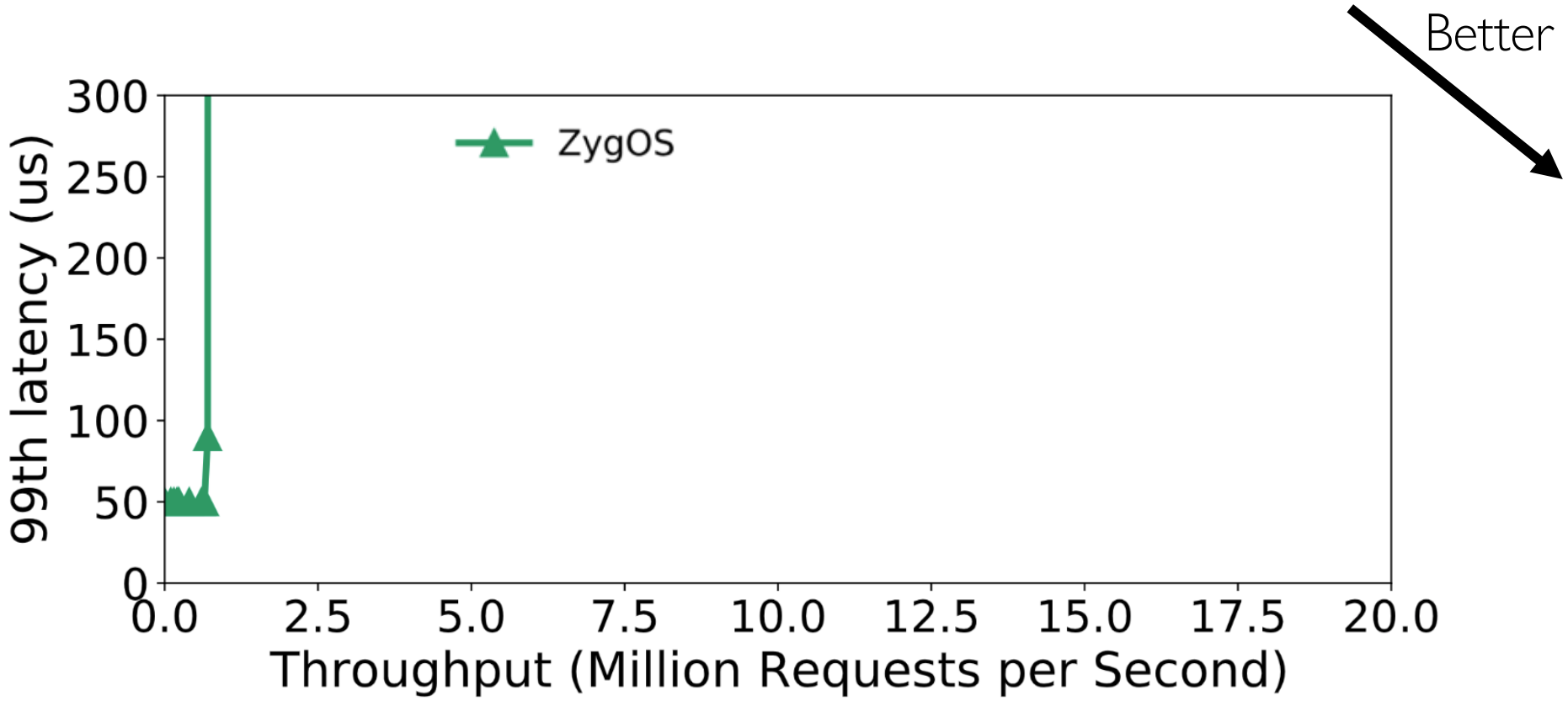
    S/W based local c-FCFS
    - comparable with S/W baselines
  - Integrated NIC +

    H/W based local c-FCFS
    - comparable with H/W baselines

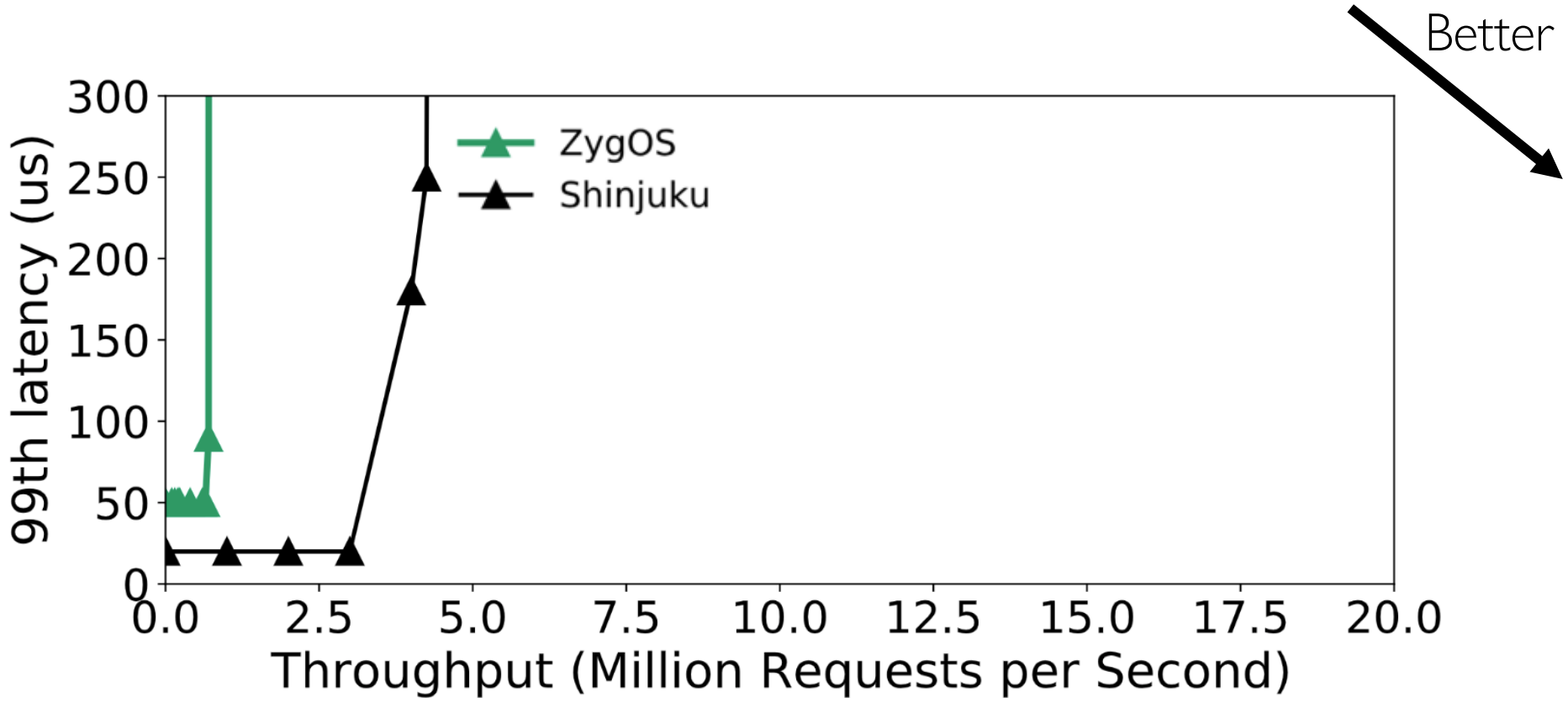
Key Evaluation: Throughput @ SLO



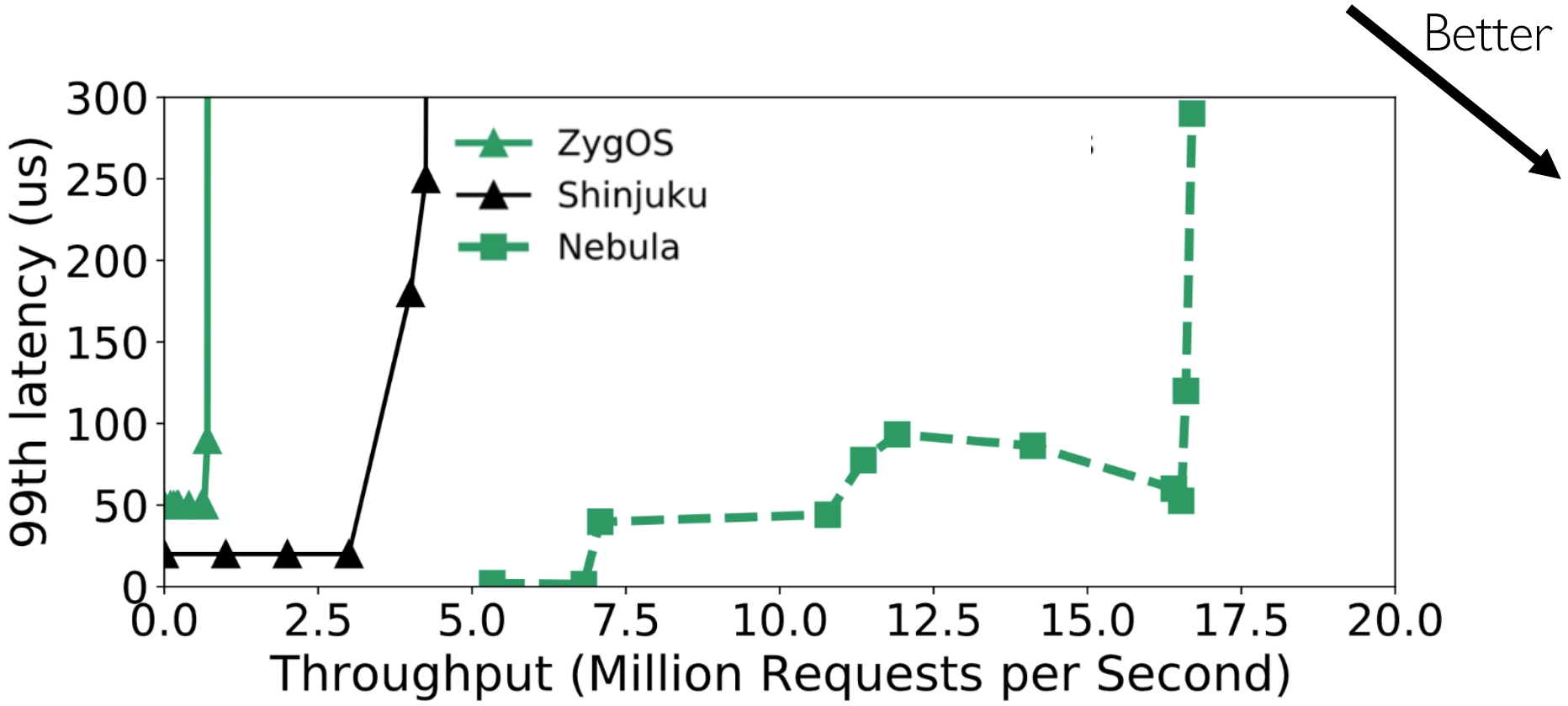
Key Evaluation: Throughput @ SLO



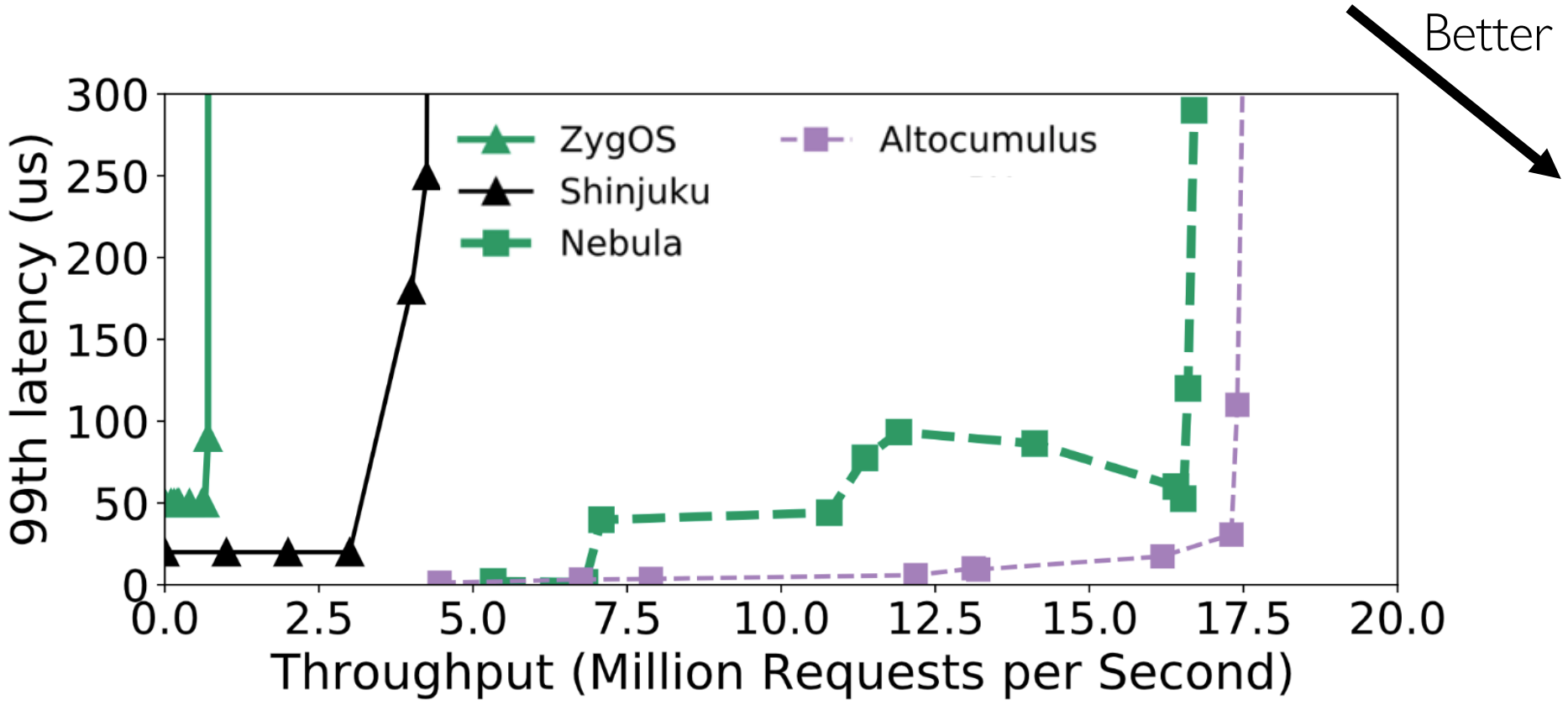
Key Evaluation: Throughput @ SLO



Key Evaluation: Throughput @ SLO

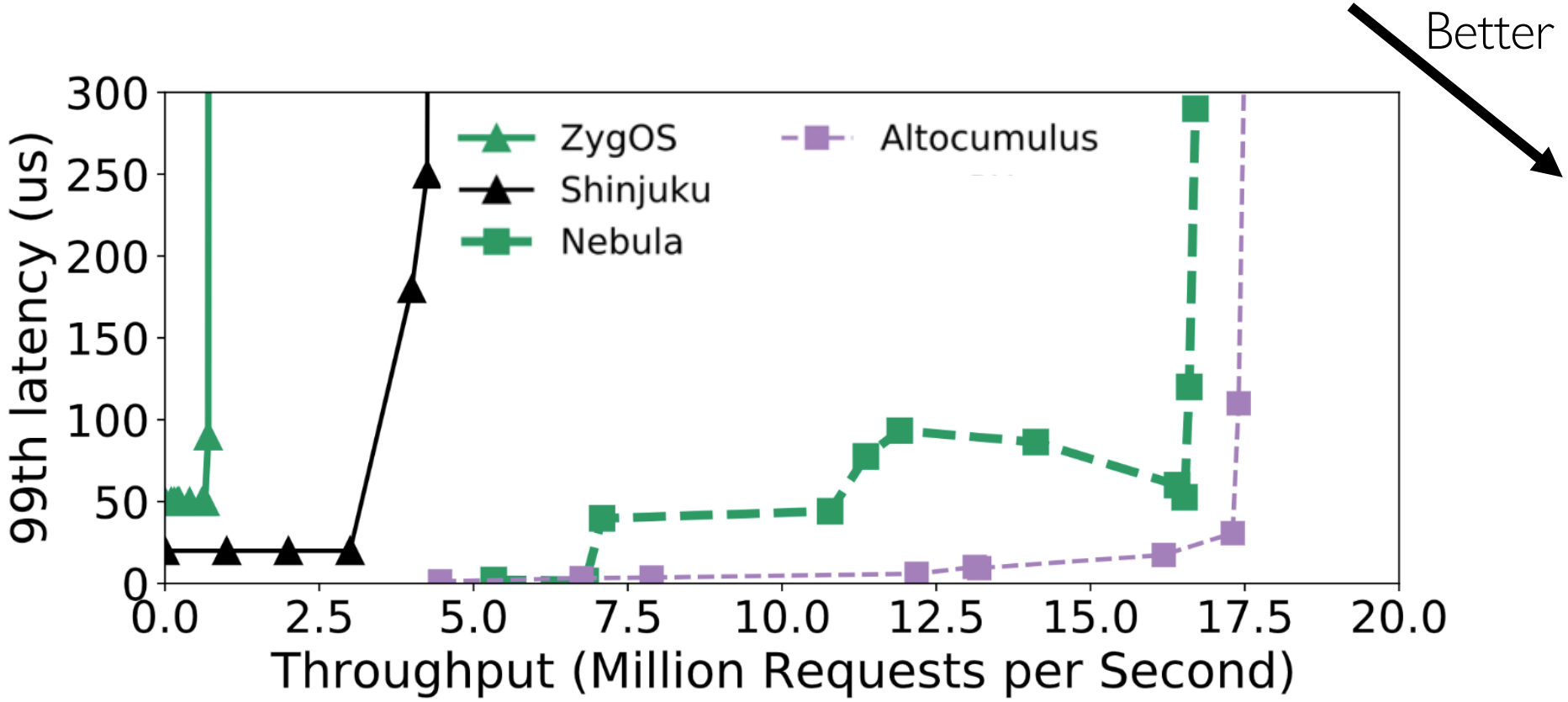


Key Evaluation: Throughput @ SLO



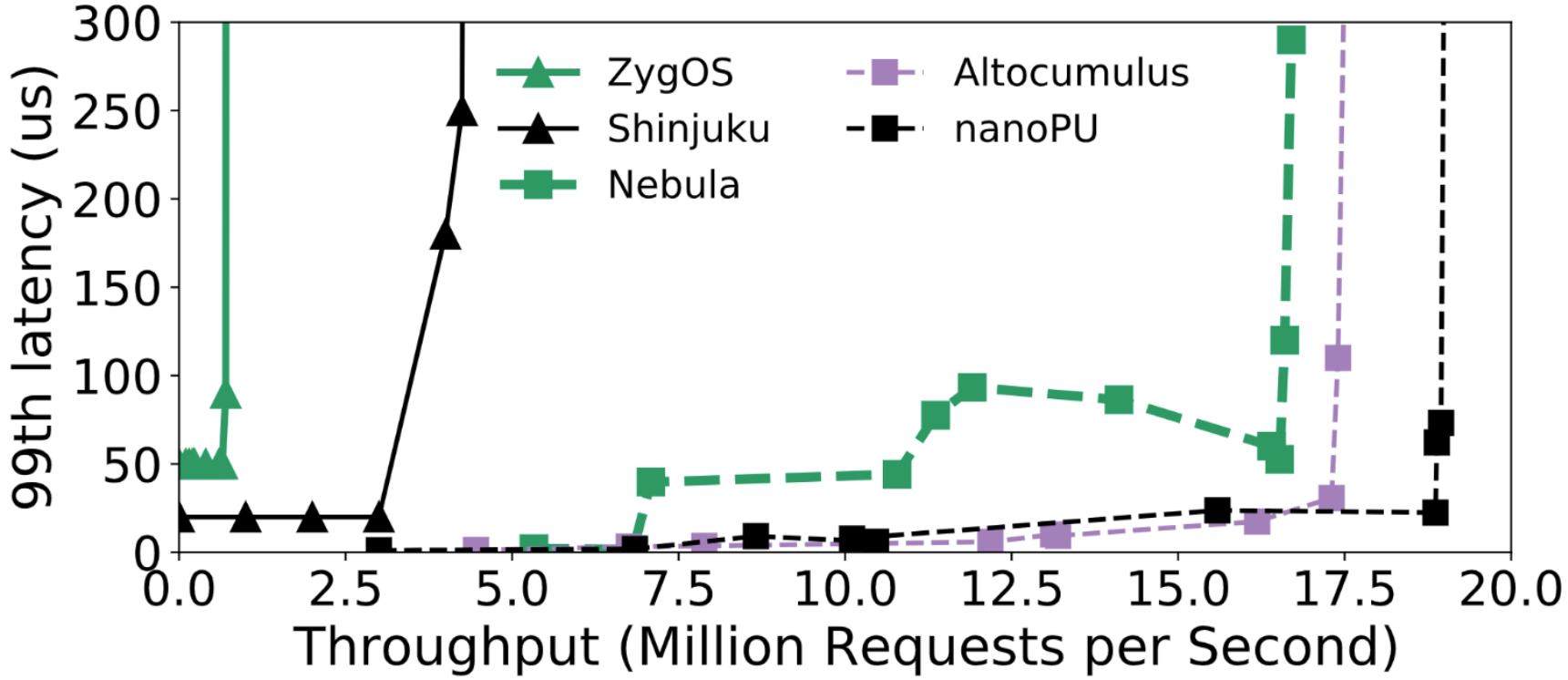
Key Evaluation: Throughput @ SLO

- Outperform S/W solutions (ZygOS, Shinjuku) by up to 24x under 10 microsecond SLO



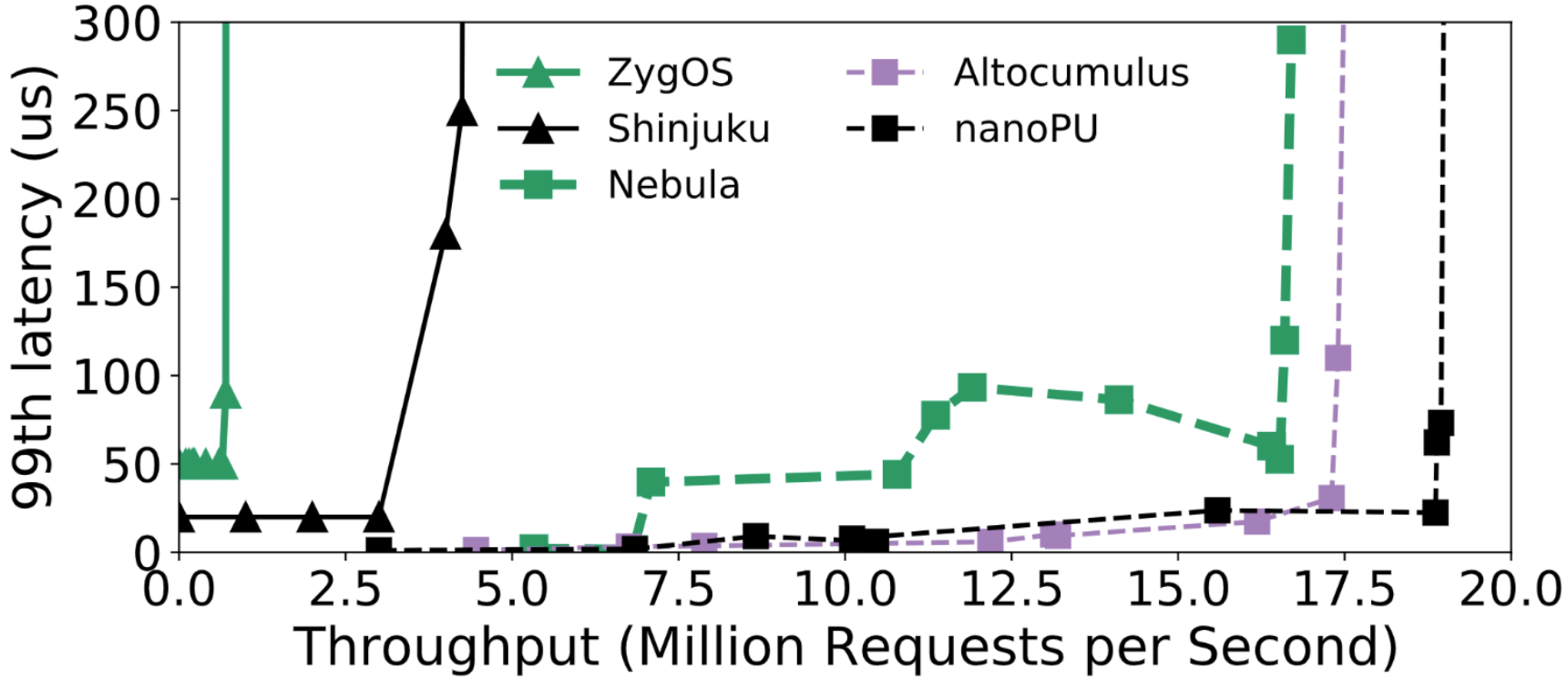
Key Evaluation: Throughput @ SLO

- Outperform S/W solutions (ZygOS, Shinjuku) by up to 24x under 10 microsecond SLO

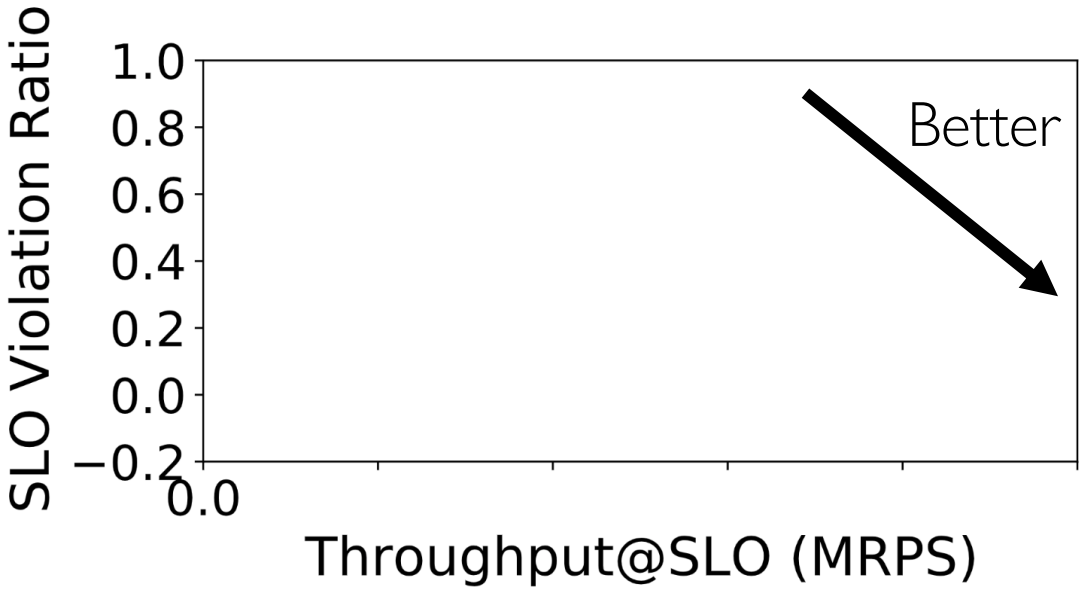


Key Evaluation: Throughput @ SLO

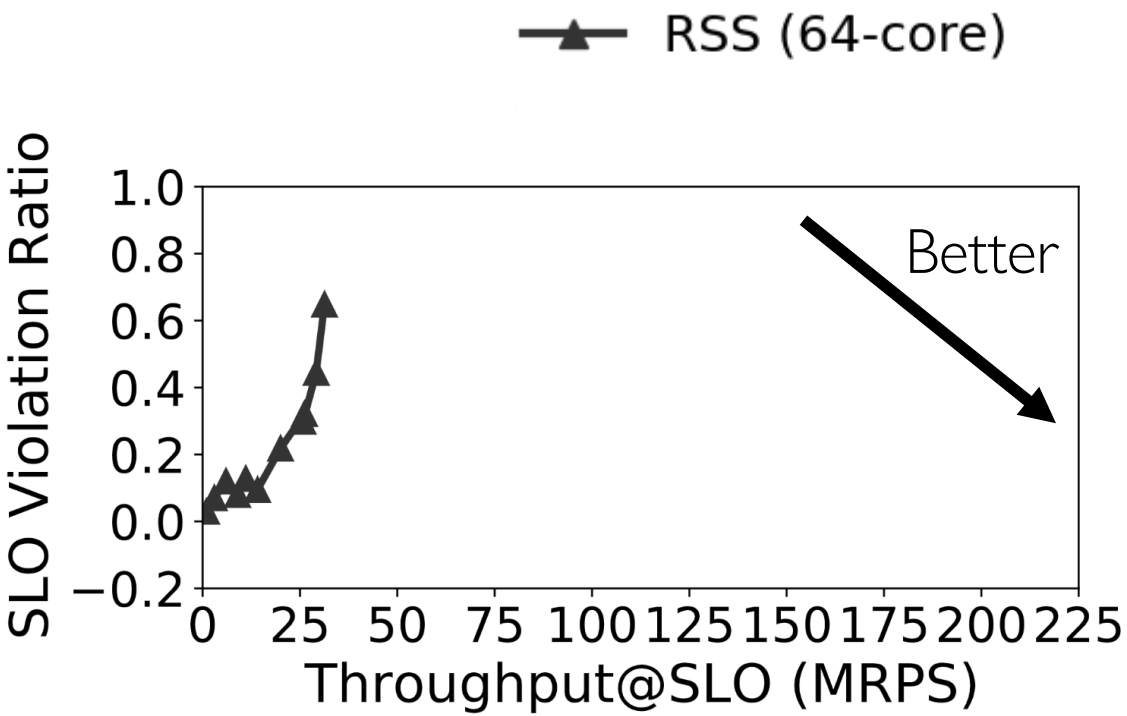
- Outperform S/W solutions (ZygOS, Shinjuku) by up to 24x under 10 microsecond SLO
- Has comparable throughput to highly-optimized H/W runtimes (Nebula & nanoPU)



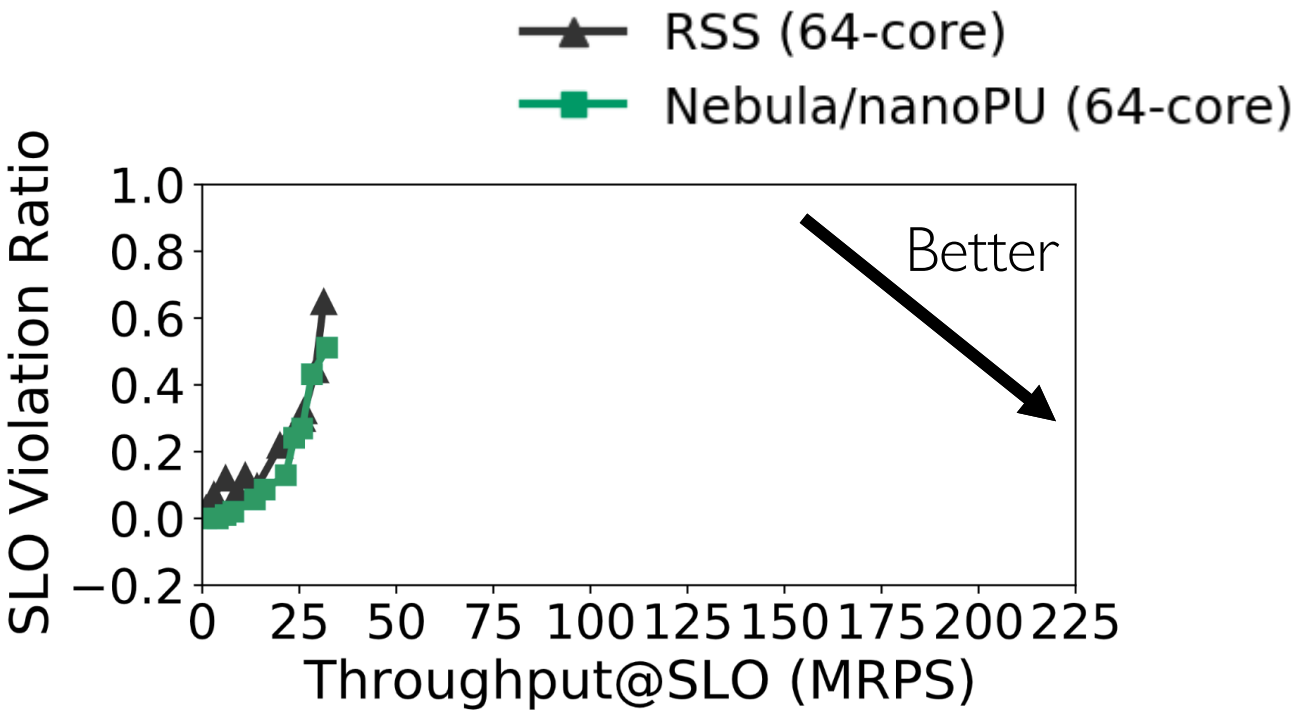
Key Evaluation: Scalability



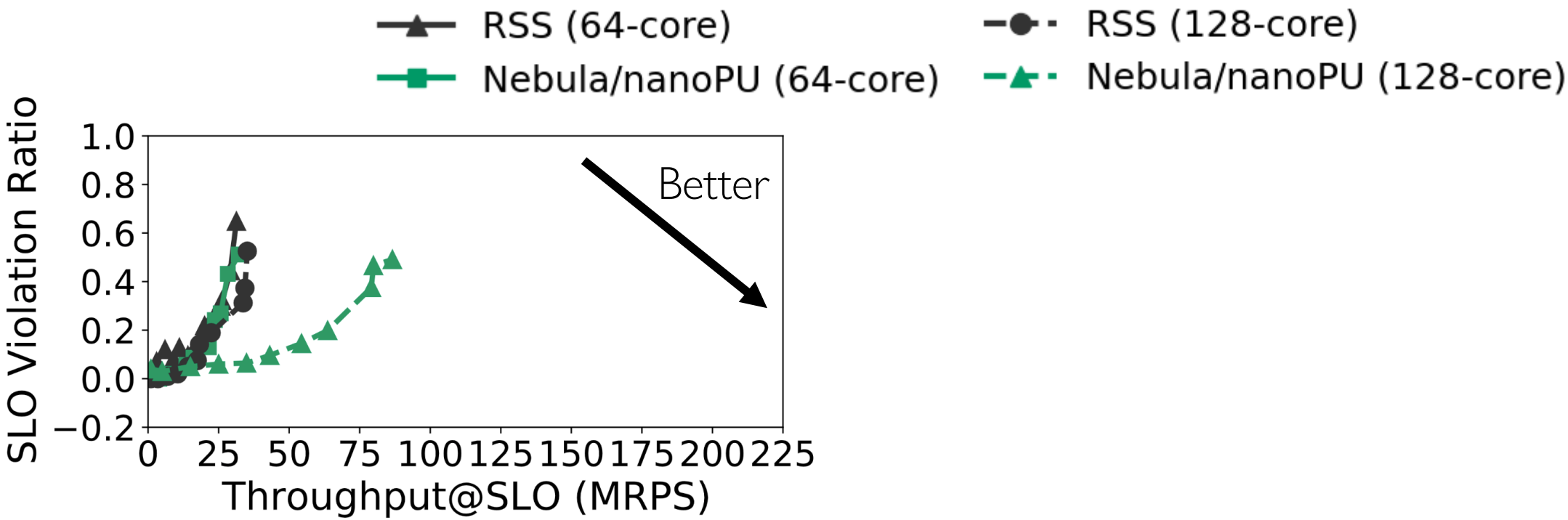
Key Evaluation: Scalability



Key Evaluation: Scalability

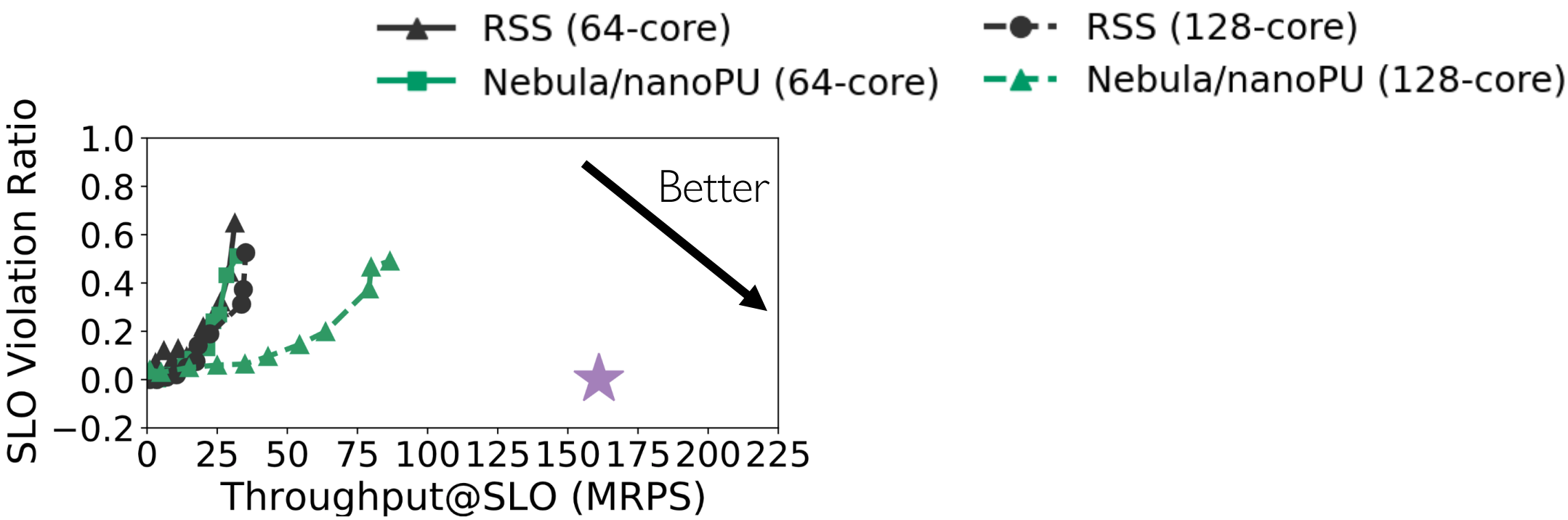


Key Evaluation: Scalability



Key Evaluation: Scalability

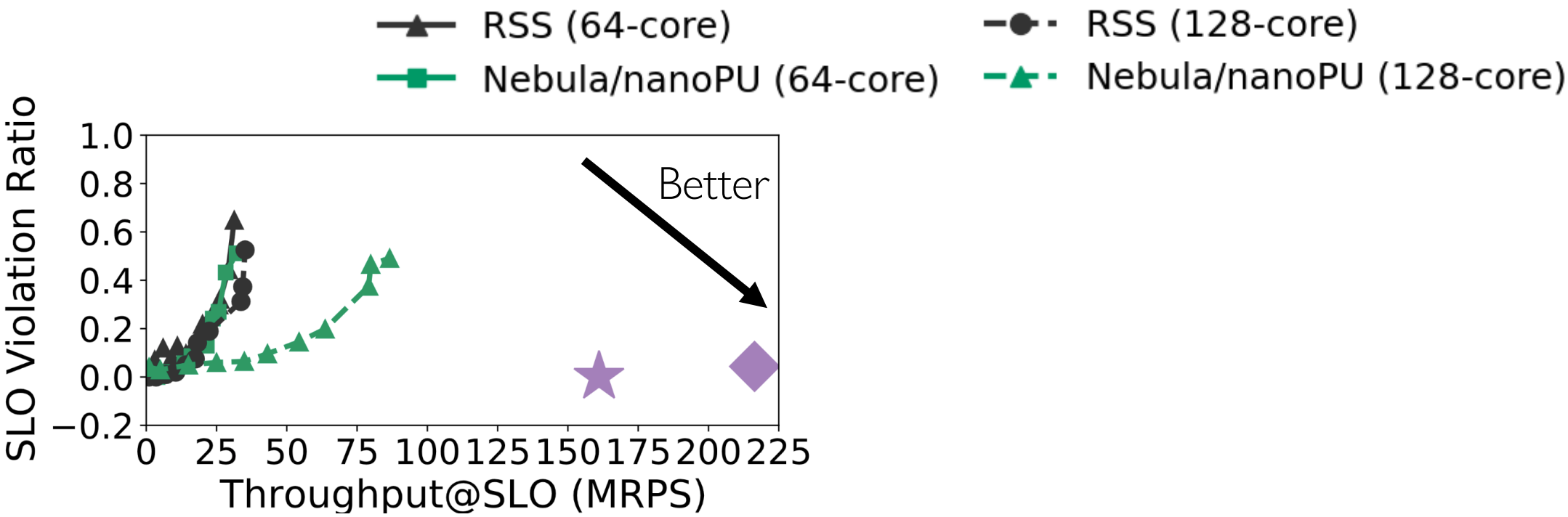
★ <0.001% SLO violation and achieve 161 MRPS, scalable to 256 cores



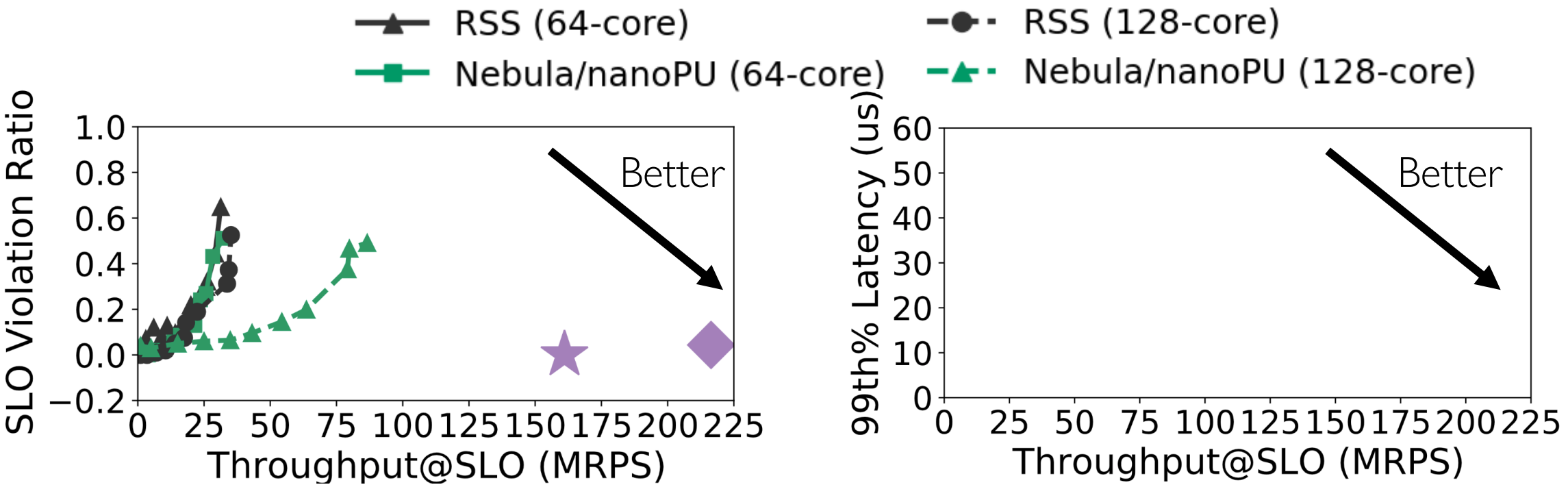
Key Evaluation: Scalability

★ <0.001% SLO violation and achieve 161 MRPS, scalable to 256 cores

◆ <5% SLO violation and achieve 216 MRPS, scalable to 256 cores

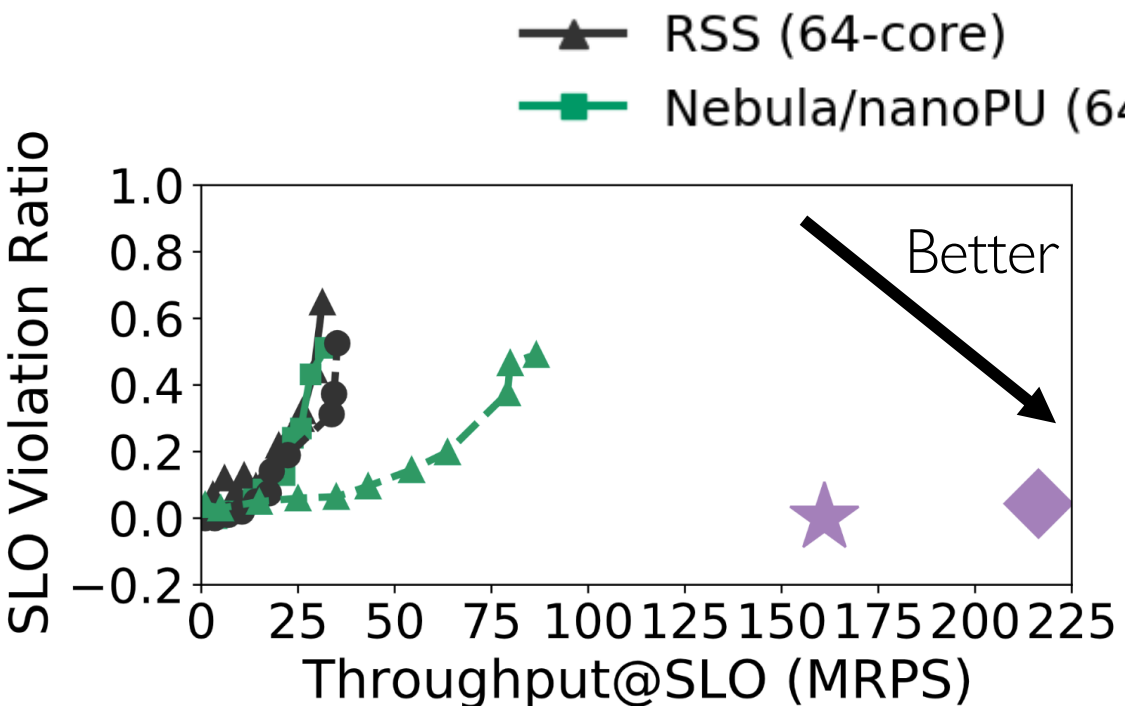


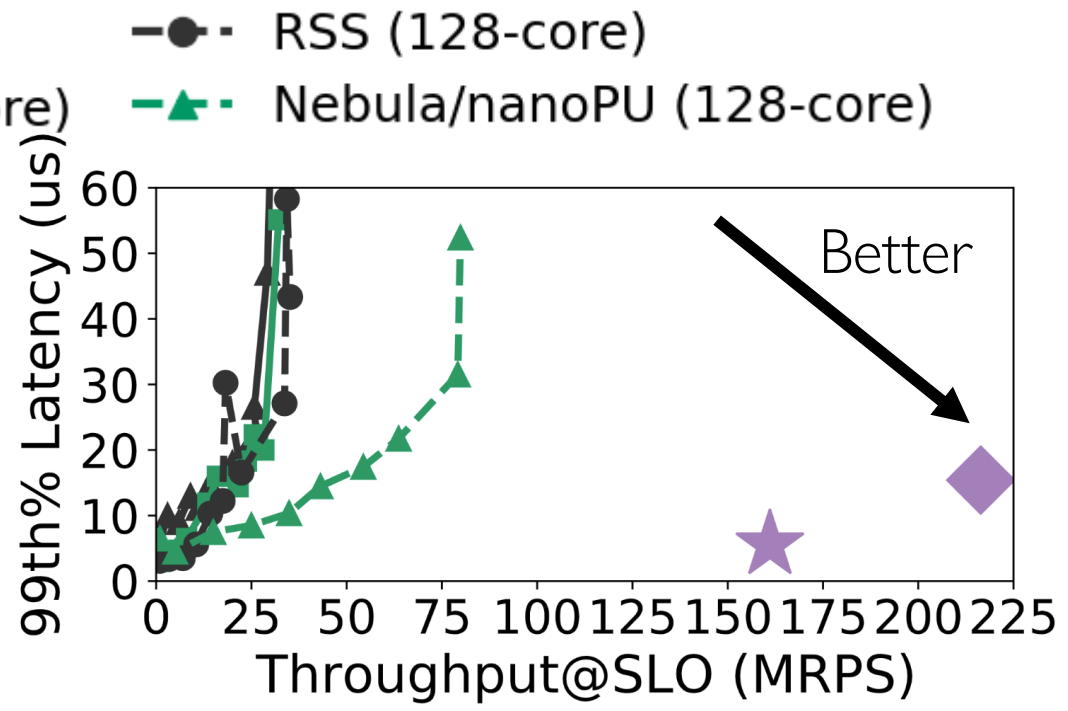
Key Evaluation: Scalability



Key Evaluation: Scalability

★ Achieve 161 MRPS while 99%th <= 5.57 us

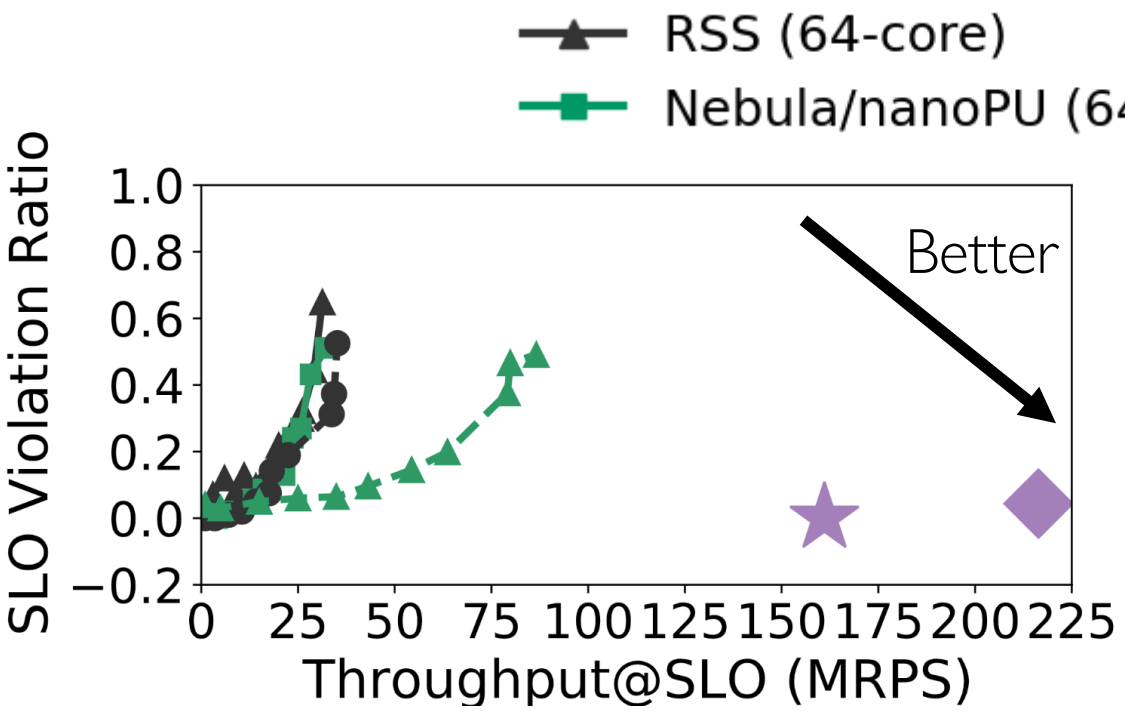


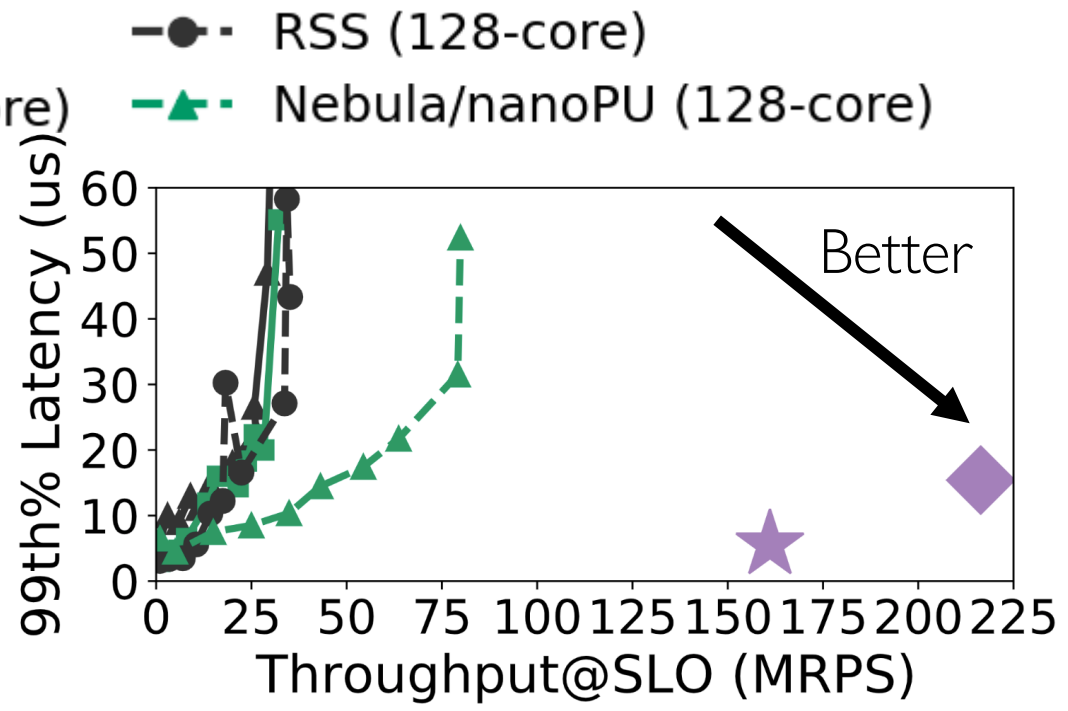


Key Evaluation: Scalability

★ Achieve 161 MRPS while 99%th <= 5.57 us

◆ Achieve 216 MRPS while 99%th <= 15.41 us

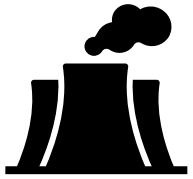




Conclusions

High volume of requests
(400+ Gbps)

Strict SLO
(99th% latency@µs)



High CPU efficiency
(64-256 cores)

Policy
Runtime
Mechanism

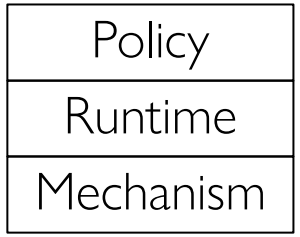
→ SLO violation prediction via queueing theory, proactive migrations

→ S/W decentralized runtime with simple H/W primitives

→ Efficient direct register messaging w/ minimal H/W overheads

Altocumulus
Scheduling Sub-system

Key Take-Aways



Altocumulus Stack

- With RPC stack components getting more optimized, other system-level components, e.g., scheduling, would become the future bottleneck
- Scheduling at **10s ns level is mandatory** for µs-scale SLO, to achieve that
  - Policy should be proactive and accurate/effective
  - Mechanism should be fast enough
- Carefully re-partitioning system stack between software or hardware can open opportunities for scalability that existing systems
  - Decentralized runtime preserves scalability
  - Software runtime offers adaptivity to various load patterns
  - Decentralized software runtime with minimal hardware overhead can reconcile design trade-offs across low tail latency, high through and high utilization (scalability)

Thank you for your attention

Jiechen Zhao, Iris Uwizeyimana, Karthik Ganesan
Mark C. Jeffrey, Natalie Enright Jerger

55th IEEE/ACM International Symposium on Microarchitecture (MICRO), Chicago, Oct. 2022
https://www.eecg.utoronto.ca/~mcj/papers/2022.altocumulus.micro.pdf