APRIL 2023, NSDI’23

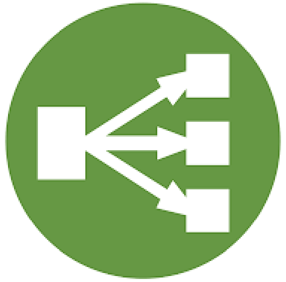
Ringleader: Efficiently Offloading Intra-Server Orchestration to NICs

Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E. Stephens,
Hassan Wassel and Aditya Akella

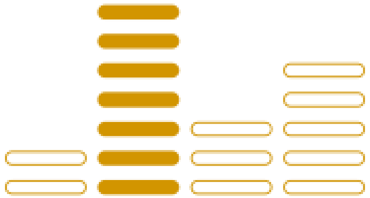
Three requirements of online cloud services

- [R1] Minimize request tail latency
  - ~10s microsecond tail latency.
- [R2] Enforce appropriate request prioritization
  - Requests have varying importance and SLO.
- [R3] Maximize CPU efficiency with interference management
  - Pack multiple applications while mitigating interference between them.

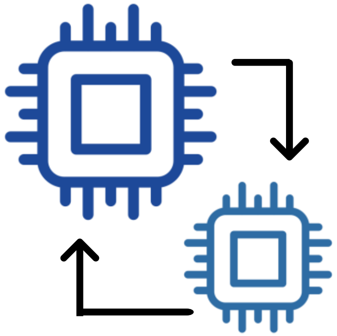
Intra-server orchestration is necessary



Load Balancing
[R1]

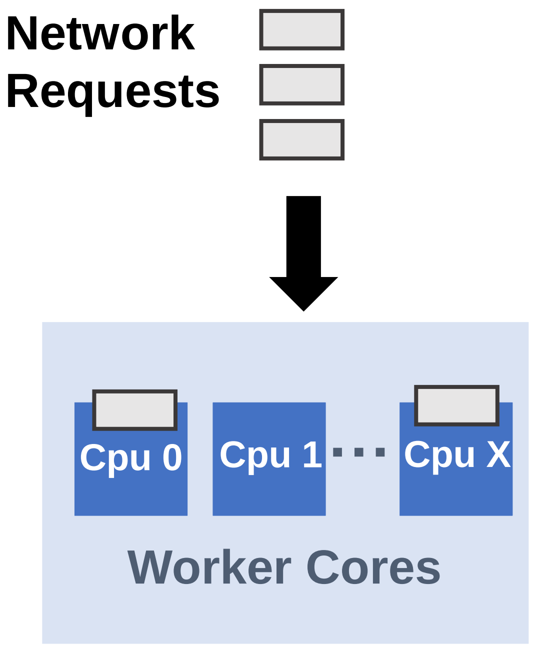


Request Scheduling
[R2]

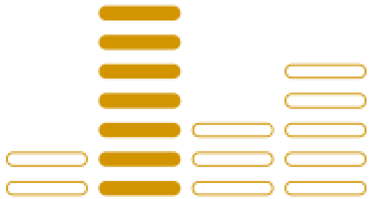


CPU Allocation
[R3]

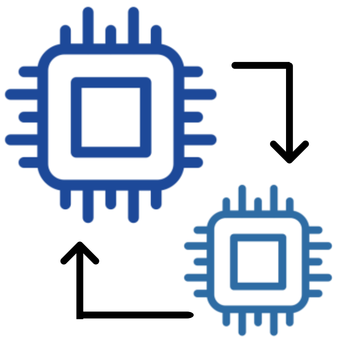
Intra-server orchestration is necessary



Load Balancing
[R1]

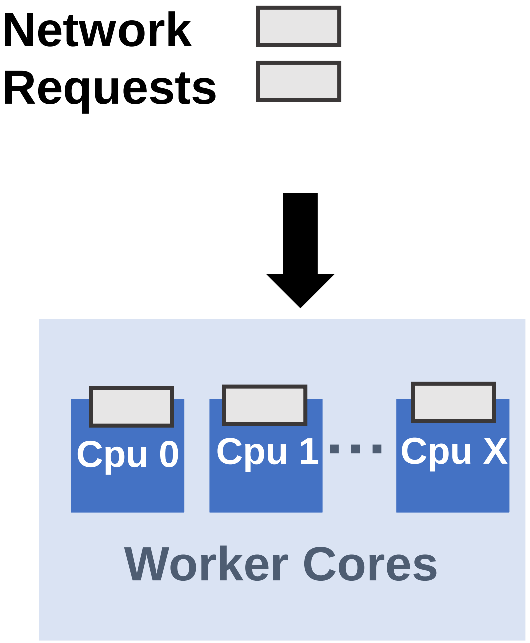


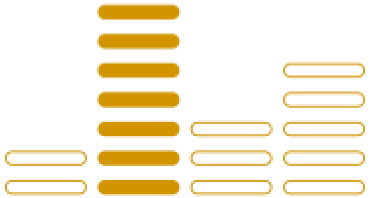
Request Scheduling
[R2]



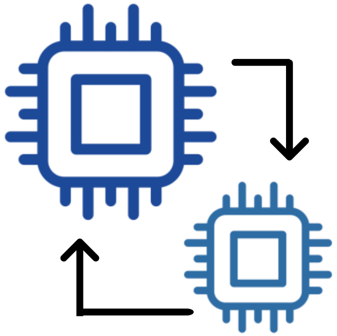
CPU Allocation
[R3]

Intra-server orchestration is necessary





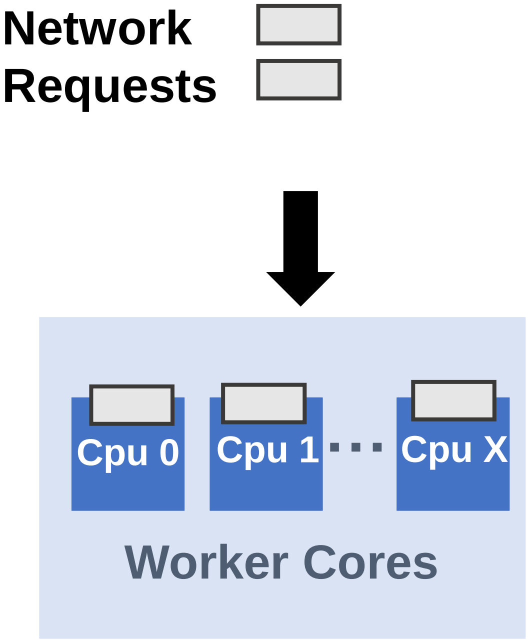
Request Scheduling
[R2]



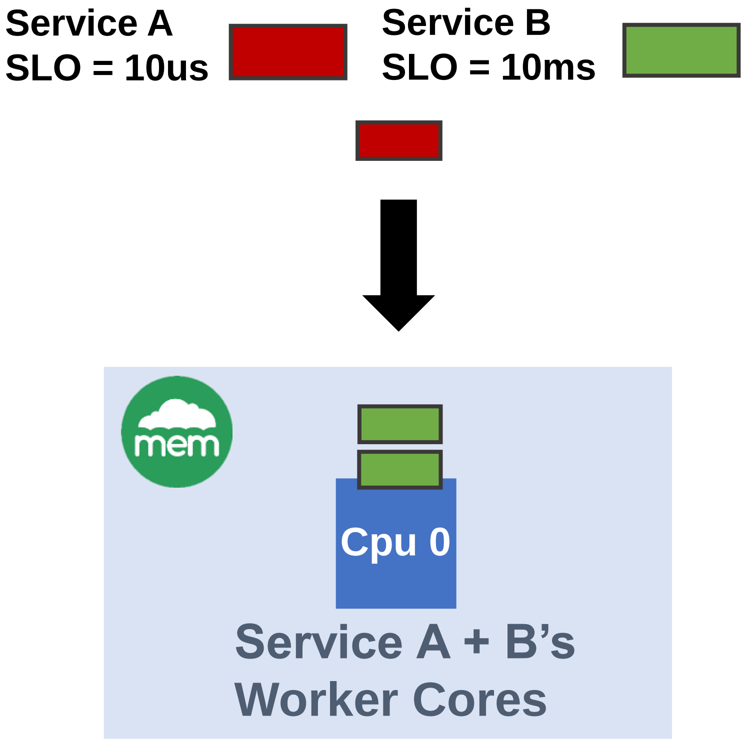
CPU Allocation
[R3]

Load Balancing
[R1]

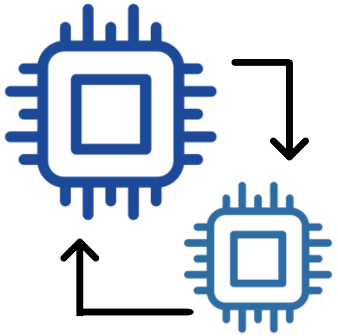
Intra-server orchestration is necessary



Load Balancing
[R1]

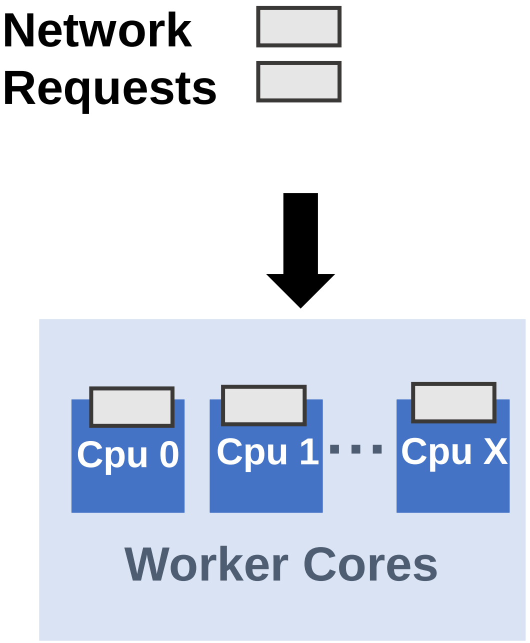


Request Scheduling
[R2]

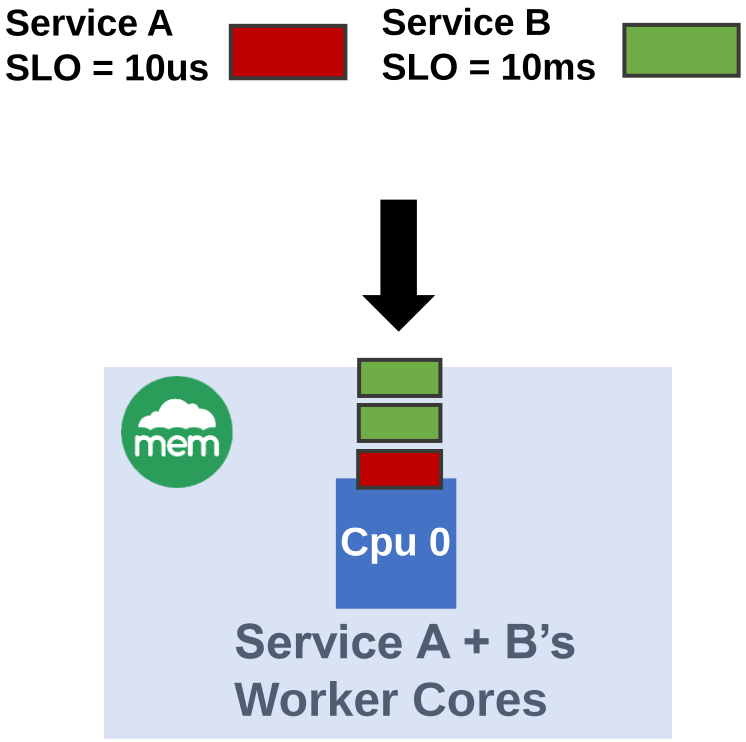


CPU Allocation
[R3]

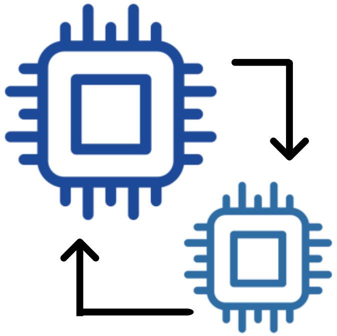
Intra-server orchestration is necessary



Load Balancing
[R1]

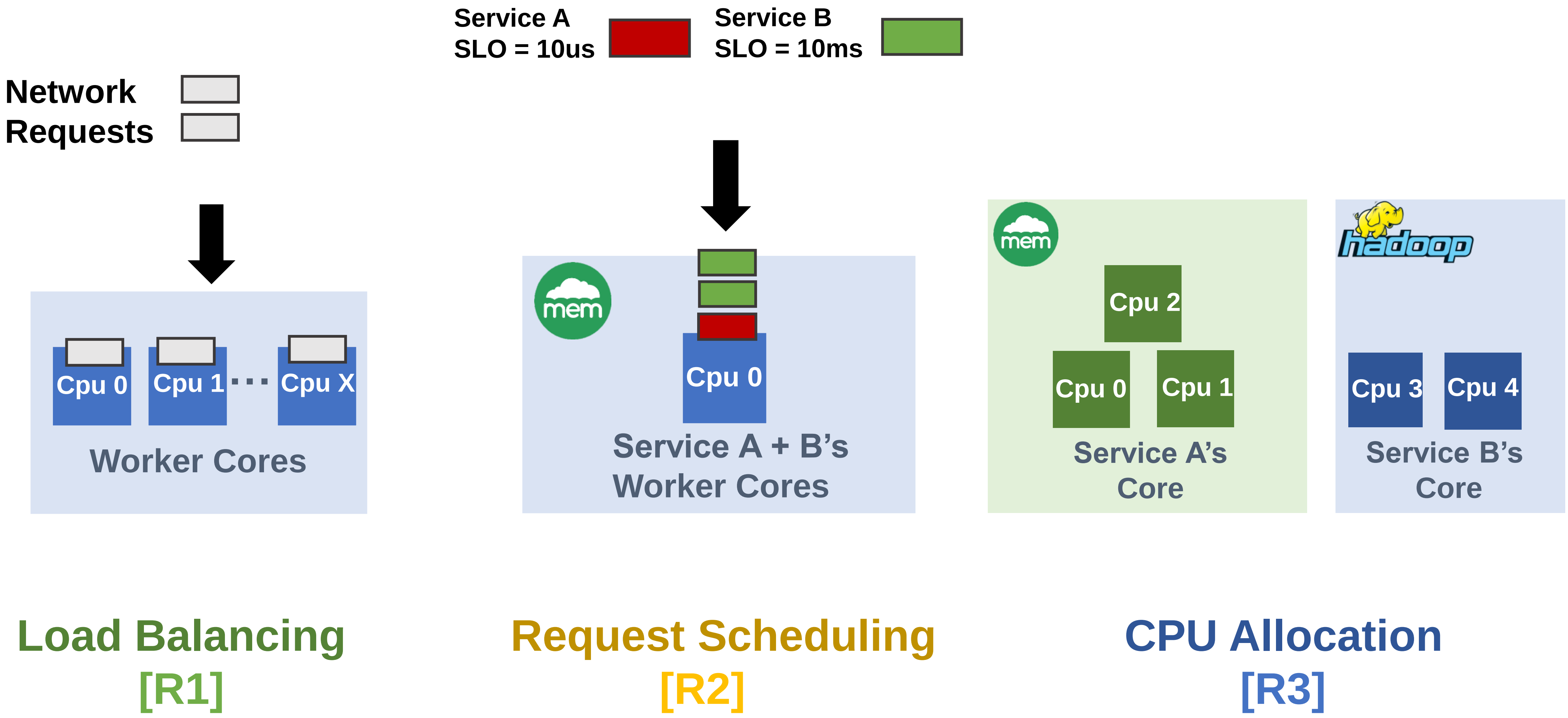


Request Scheduling
[R2]

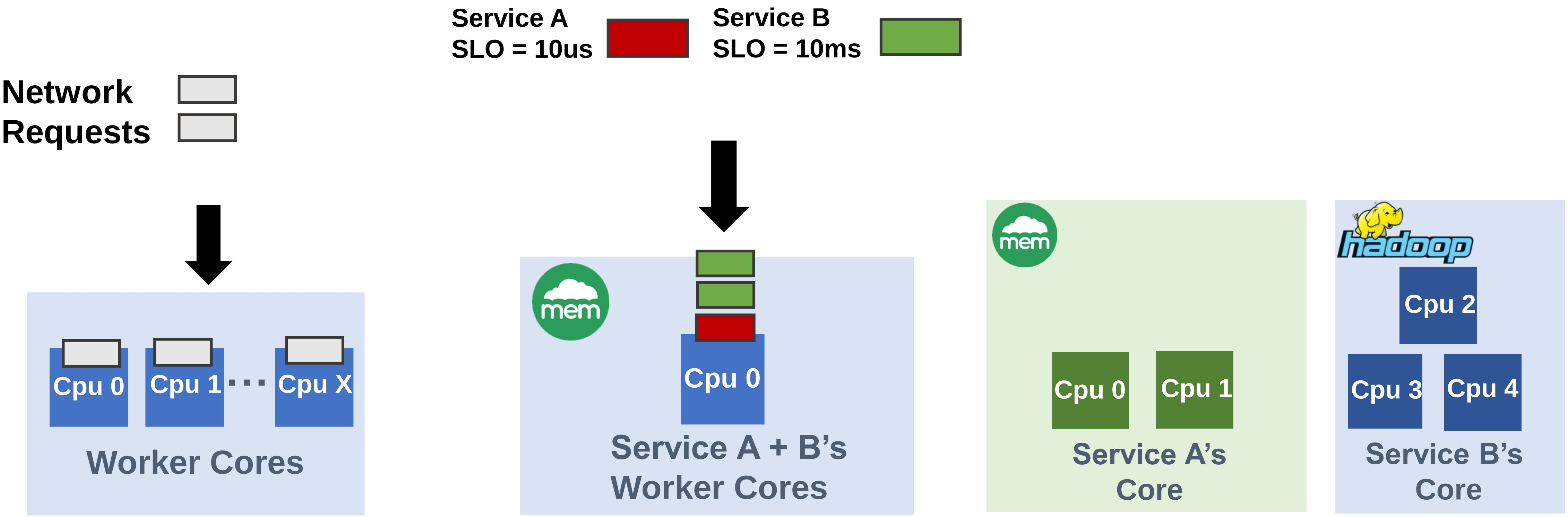


CPU Allocation
[R3]

Intra-server orchestration is necessary



Intra-server orchestration is necessary



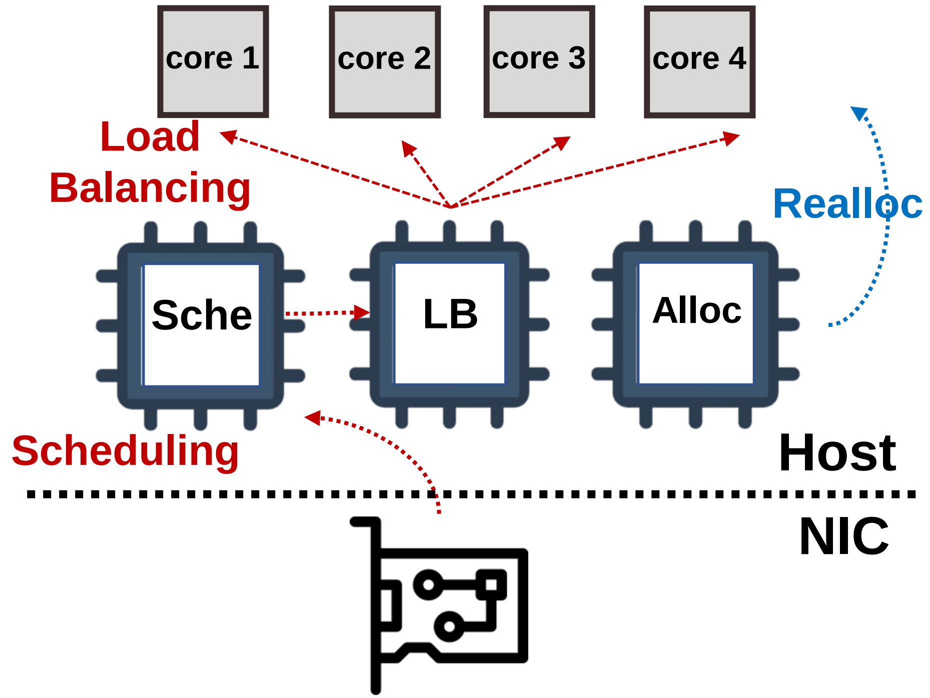
Load Balancing
[R1]

Request Scheduling
[R2]

CPU Allocation
[R3]

Intra-server Orchestration Today

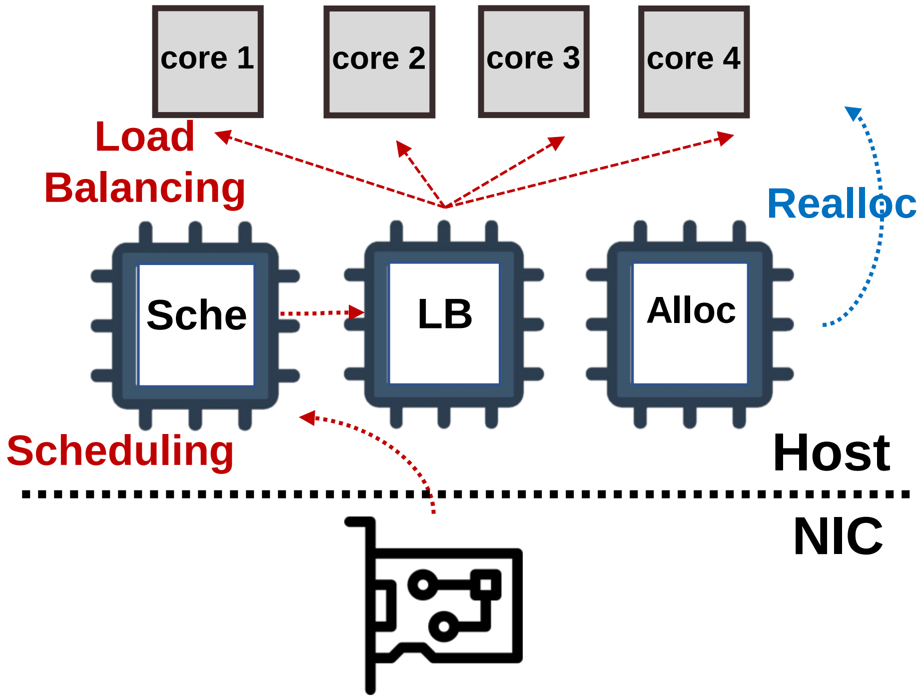
Use centralzied CPU cores to make the centralized scheduling/load balancing/CPU allocation decisions *[Shinjuku@NSDI'19, Caladan@OSDI'20]*.



Intra-server Orchestration Today

Use centralzied CPU cores to make the centralized scheduling/load balancing/CPU allocation decisions *[Shinjuku@NSDI'19, Caladan@OSDI'20]*.

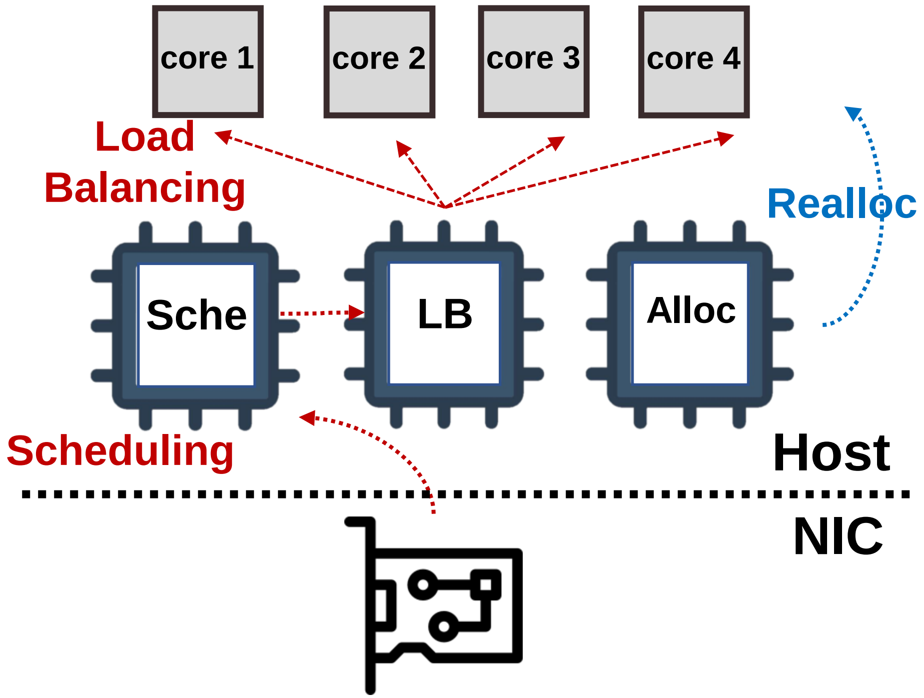
- **Advantage:**
  - The centralized approach provides optimal performance.



Intra-server Orchestration Today

Use centralzied CPU cores to make the centralized scheduling/load balancing/CPU allocation decisions *[Shinjuku@NSDI'19, Caladan@OSDI'20]*.

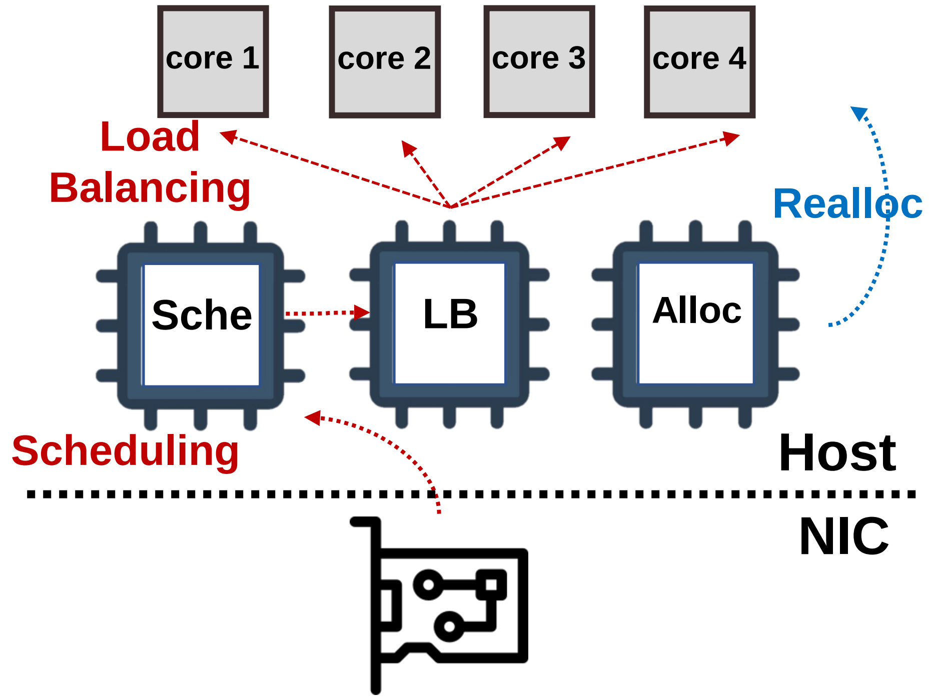
- **Advantage:**
  - The centralized approach provides optimal performance.
- **Problems:**
  - Wasted cores.
  - Limited scalability.



Intra-server Orchestration Today

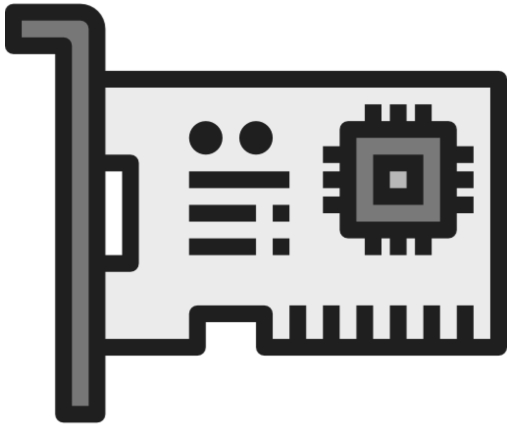
Use centralzied CPU cores to make the centralized scheduling/load balancing/CPU allocation decisions *[Shinjuku@NSDI'19, Caladan@OSDI'20]*.

- **Advantage:**
  - The centralized approach provides optimal performance.
- **Problems:**
  - Wasted cores.
  - Limited scalability.



Is it possible to achieve **scalable centralized** intra-server orchestration with **minimal CPU overhead**?

NIC-driven hardware orchestration

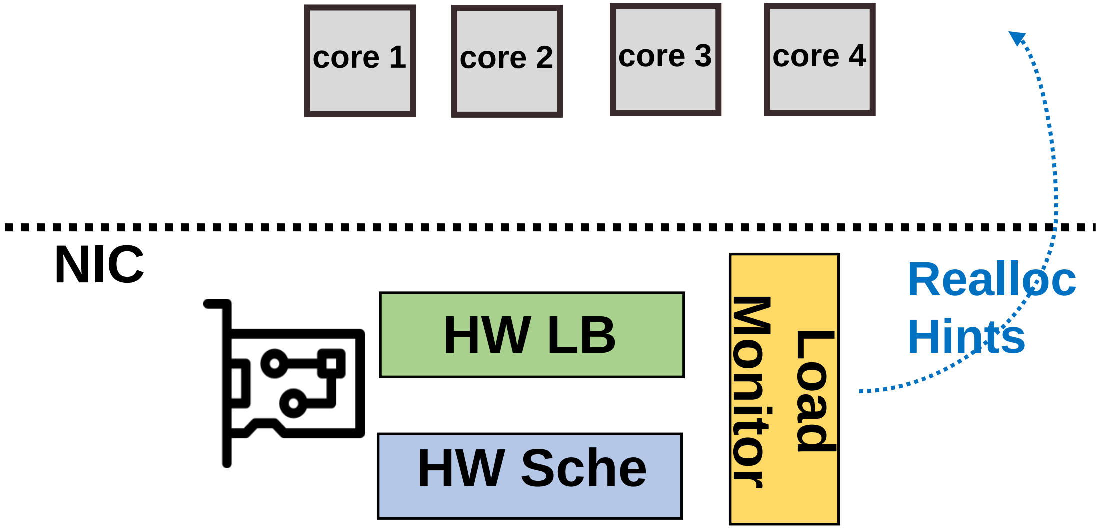


Modern NICs offer three opportunities:

- **Centralized:** All network requests must pass through the NIC.
- **Scalability:** NIC accelerators can be designed to operate at line rate.
- **Minimal Host CPU Overhead:** Offloading frees up host cores.

Ringleader Overview:

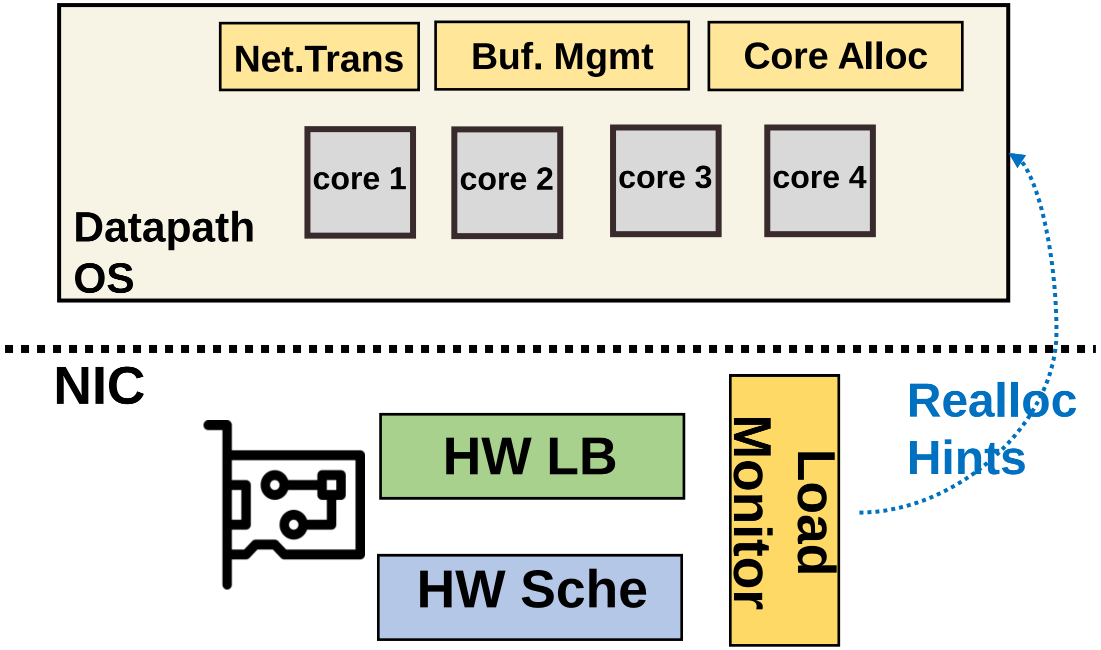
- Ringleader is a new **NIC architecture** that utilizes novel hardware offloads to perform centralized orchestration.
    - Load balancing offload.
    - Scheduling offload.
    - NIC-assisted CPU allocation.



Ringleader Overview:

- Ringleader is a new **NIC architecture** that utilizes novel hardware offloads to perform centralized orchestration.
    - Load balancing offload.
    - Scheduling offload.
    - NIC-assisted CPU allocation.

- The host uses a Datapath OS to manage services.

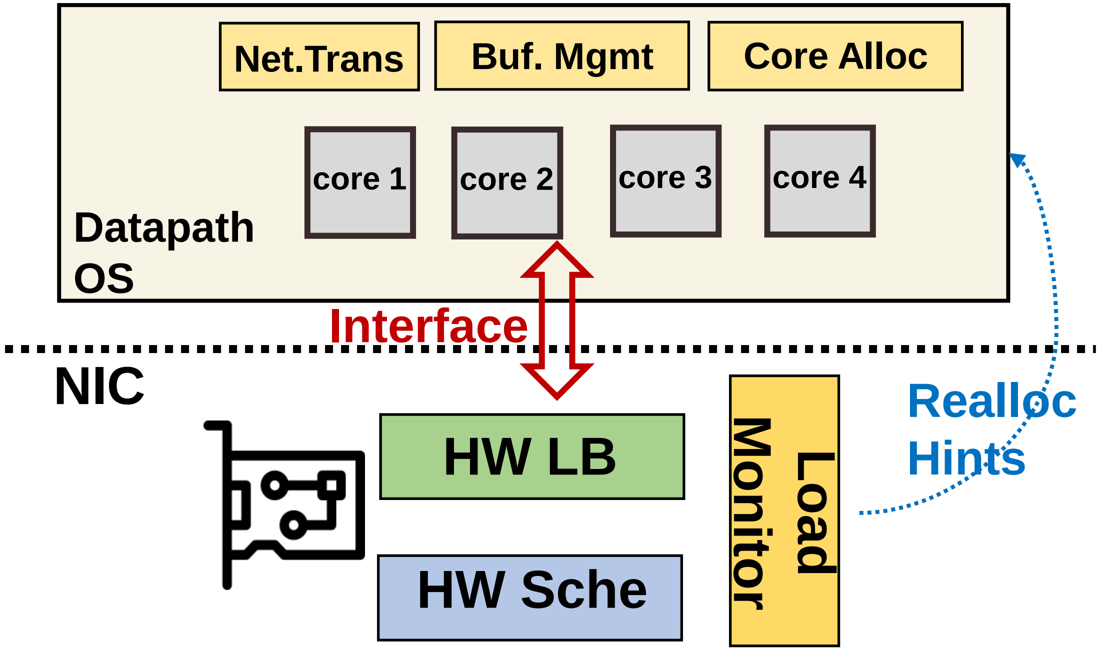


Ringleader Overview:

- Ringleader is a new **NIC architecture** that utilizes novel hardware offloads to perform centralized orchestration.
  - Load balancing offload.
  - Scheduling offload.
  - NIC-assisted CPU allocation.

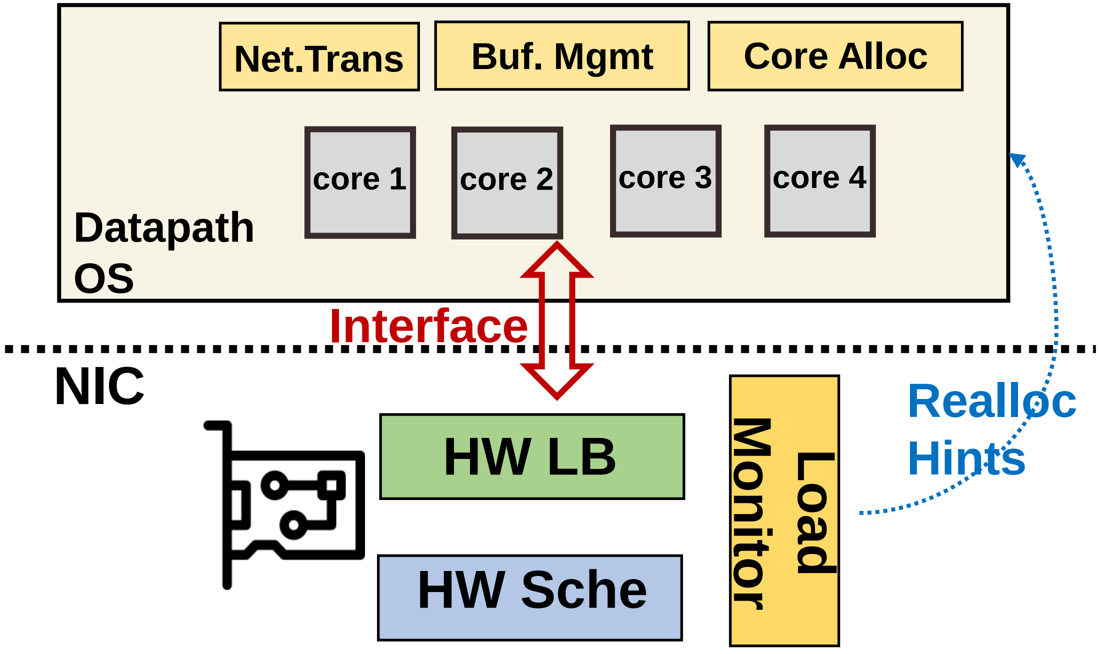
- The host uses a Datapath OS to manage services.

- A new Datapath OS-NIC interface.



Ringleader Overview:

- Ringleader is a new **NIC architecture** that utilizes novel hardware offloads to perform centralized orchestration.
  - Load balancing offload.
  - Scheduling offload.
  - NIC-assisted CPU allocation.
- The host uses a Datapath OS to manage services.
- A new Datapath OS-NIC interface.



Design questions of offloading scheduling and load balancing

Q1: What should be the division of labor between the host and NIC?

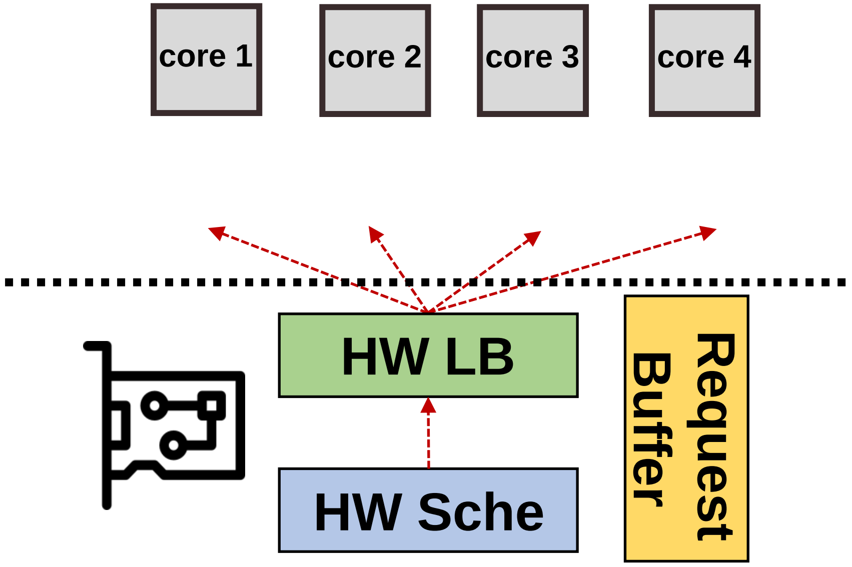
Q2: How to coordinate orchestration between the NIC and host components?

Q3: How to design the hardware to achieve efficient and high-performance offload?

Q1: Division of labor between the host and NIC

A naïve way: offload all aspects onto the NIC hardware.

- Centralized on-NIC request buffer.



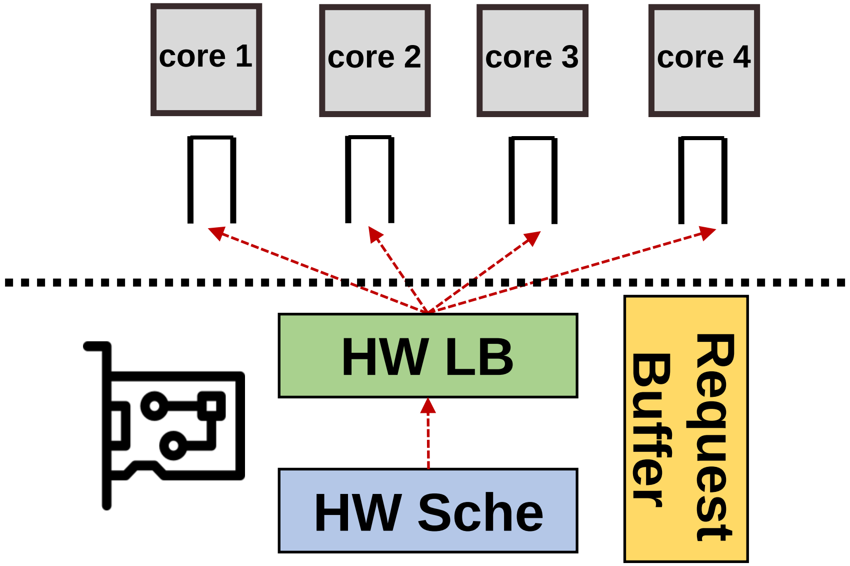
Q1: Division of labor between the host and NIC

A naïve way: offload all aspects onto the NIC hardware.

- Centralized on-NIC request buffer.

Given the PCIe delay:

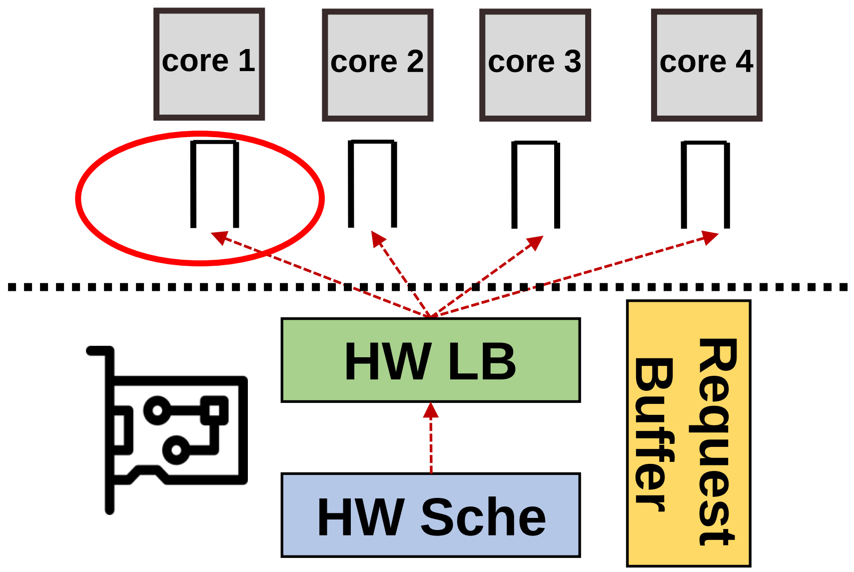
- Small per-core buffer to hide the PCIe latency.



Q1: Division of labor between the host and NIC

A naïve way: offload all aspects onto the NIC hardware.

- Centralized on-NIC request buffer.



Given the PCIe delay:

- Small per-core buffer to hide the PCIe latency.
- **Problem:** HoL blocking inside the per-core buffer.

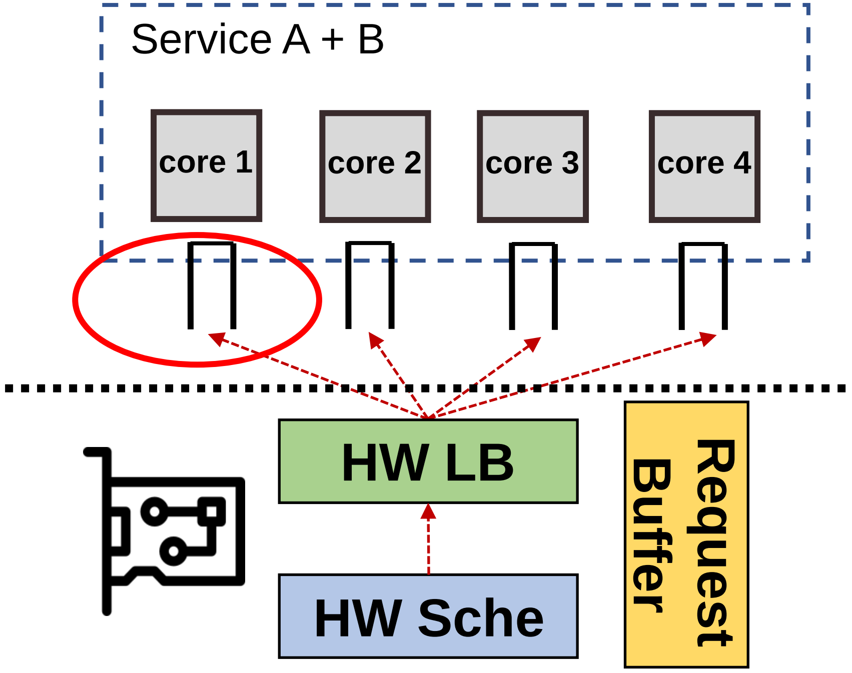
Q1: Division of labor between the host and NIC

A naïve way: offload all aspects onto the NIC hardware.

- Centralized on-NIC request buffer.

Given the PCIe delay:

- Small per-core buffer to hide the PCIe latency.
- **Problem:** HoL blocking inside the per-core buffer.



Service A's Prio = Hi
Service B's Prio = Lo

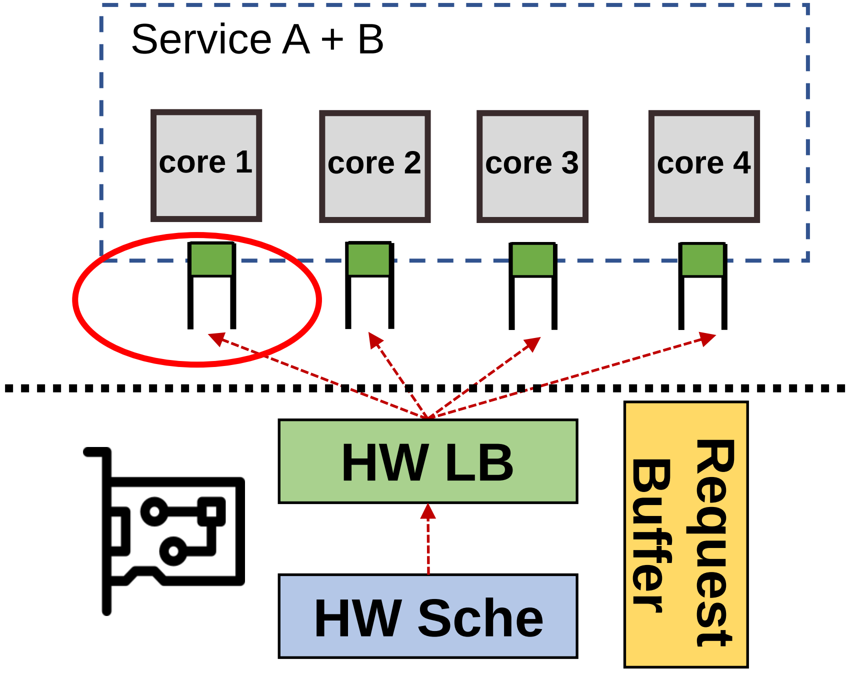
Q1: Division of labor between the host and NIC

A naïve way: offload all aspects onto the NIC hardware.

- Centralized on-NIC request buffer.

Given the PCIe delay:

- Small per-core buffer to hide the PCIe latency.
- **Problem:** HoL blocking inside the per-core buffer.



Service A's Prio = Hi
Service B's Prio = Lo

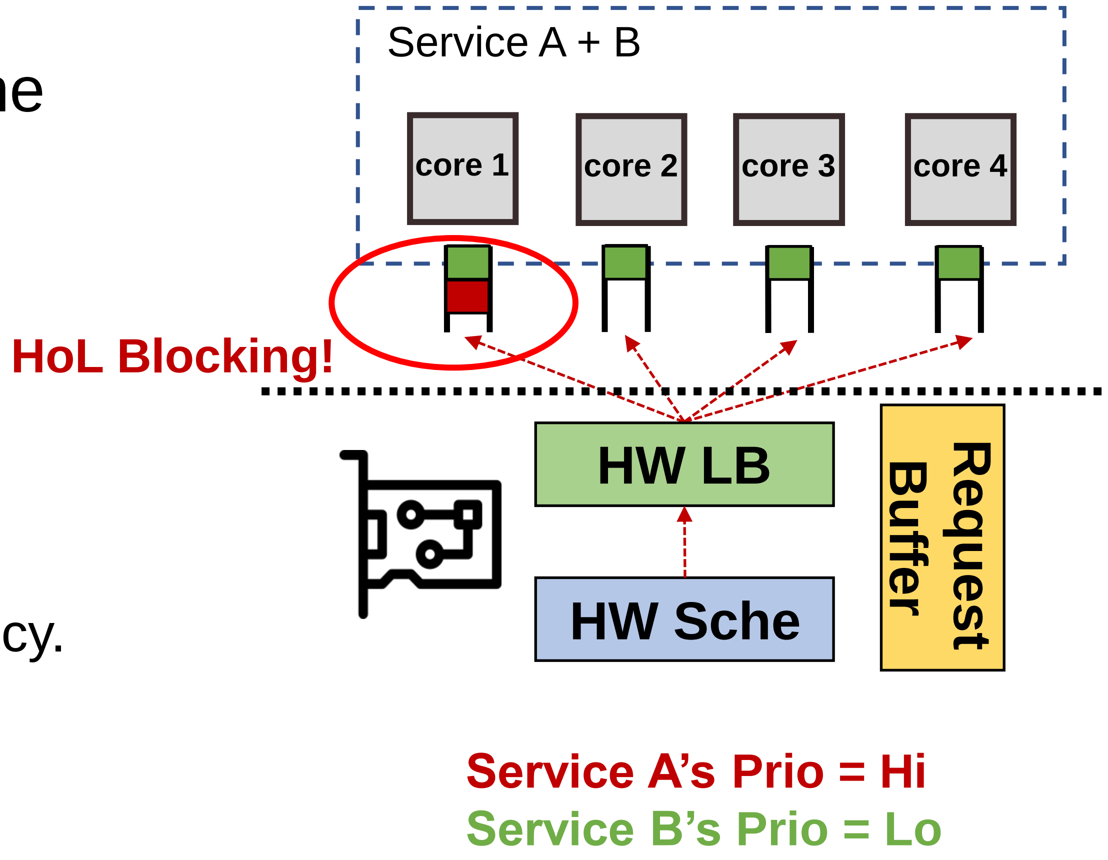
Q1: Division of labor between the host and NIC

A naïve way: offload all aspects onto the NIC hardware.

- Centralized on-NIC request buffer.

Given the PCIe delay:

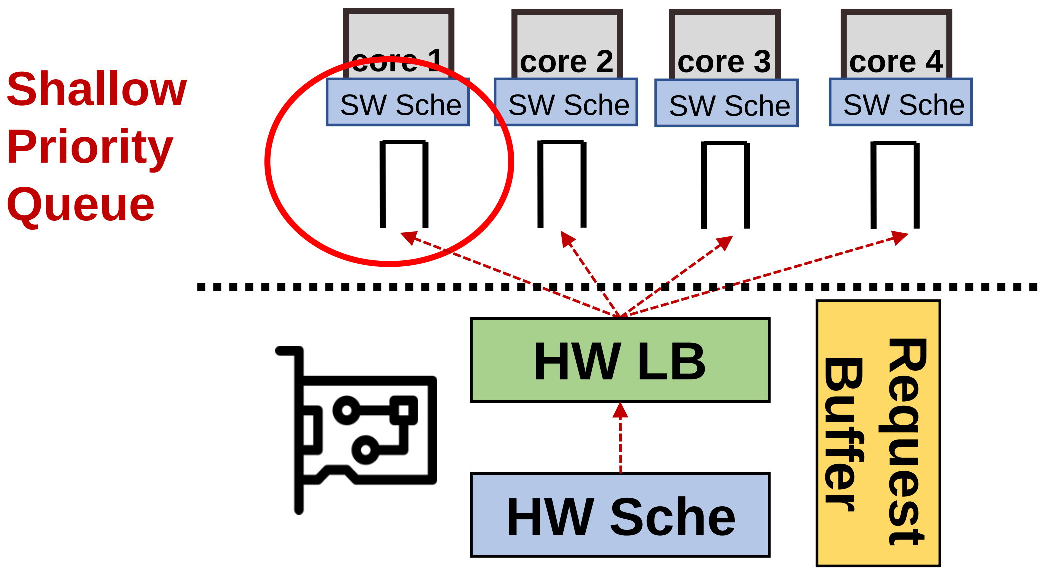
- Small per-core buffer to hide the PCIe latency.
- **Problem:** HoL blocking inside the per-core buffer.



Solution: Divide the scheduling function

Onload part of the scheduling function into host cores using shallow priority queues.

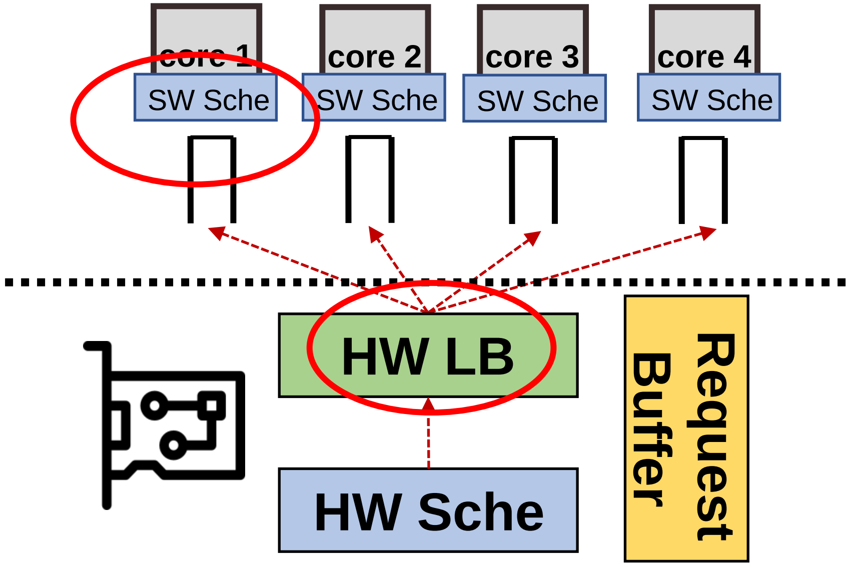
- **Priority** queue
- **Shallow** queue



Q2: Coordination between the software scheduler and the NIC load balancer

A naïve load balancer: Join-Bounded-Shortest-Queue *[nanoPU@OSDI’21, Racksched @OSDI’20]*

- JBSQ(N) steers to the core which has the minimal queue length, and each host queue has a maximum depth of N packets.

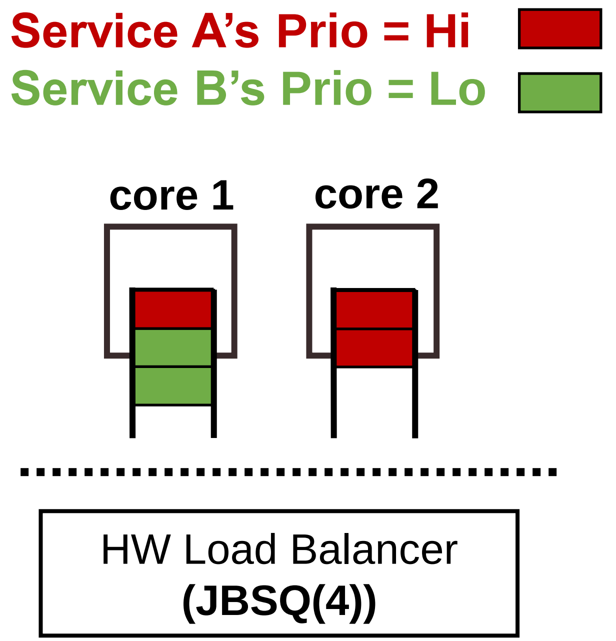


Problem: JBSQ fails because it ignores the software scheduler’s behavior!

Q2: Coordination between the software scheduler and the NIC load balancer

A naïve load balancer: Join-Bounded-Shortest-Queue *[nanoPU@OSDI'21, Racksched @OSDI'20]*

- JBSQ(N) steers to the core which has the minimal queue length, and each host queue has a maximum depth of N packets.

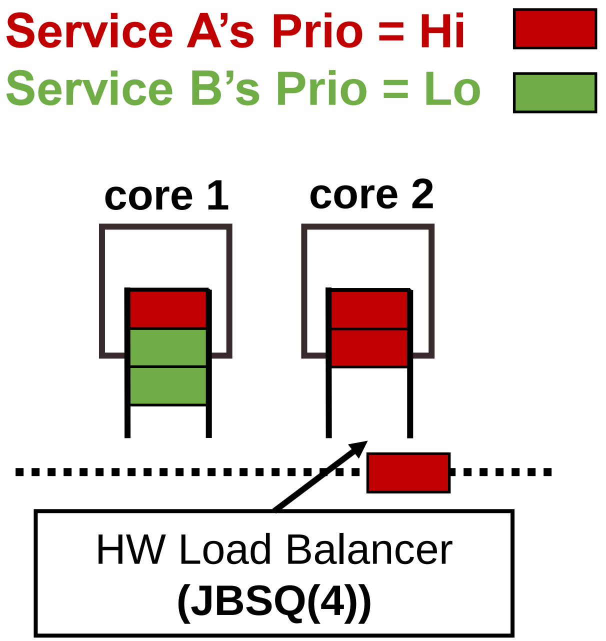


Problem: JBSQ fails because it ignores the software scheduler's behavior!

Q2: Coordination between the software scheduler and the NIC load balancer

A naïve load balancer: Join-Bounded-Shortest-Queue *[nanoPU@OSDI'21, Racksched @OSDI'20]*

- JBSQ(N) steers to the core which has the minimal queue length, and each host queue has a maximum depth of N packets.



Problem: JBSQ fails because it ignores the software scheduler's behavior!

Load Balancing with Join-Bounded-Smallest-Rank-Queue:

JBSRQ(N): steer to the core which has the minimal **rank**, and each host queue has a maximum **rank of N**.

$$Rank[A].coreC = \underbrace{\sum_{X.pri \geq A.pri} Queue[X].coreC}_{\text{Insight 1}} + \lambda * \sum_{X.pri < A.pri} Queue[X].coreC$$

[Insight 1] rank is contributed by same/higher priority requests.

Load Balancing with Join-Bounded-Smallest-Rank-Queue:

JBSRQ(N): steer to the core which has the minimal **rank**, and each host queue has a maximum **rank of N**.

λ is a constant factor between 0 and 1.

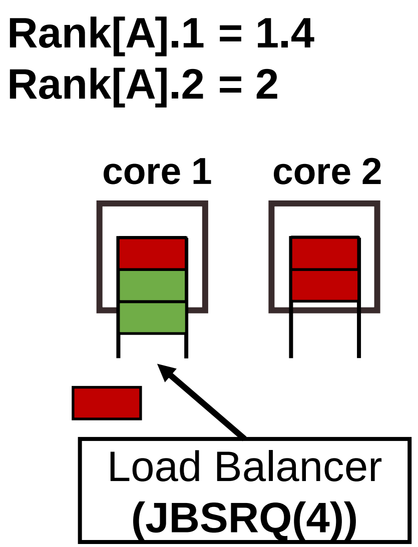
$$Rank[A].coreC = \overbrace{\sum_{X.pri \geq A.pri} Queue[X].coreC}^{\text{Insight 1}} + \overbrace{\lambda * \sum_{X.pri < A.pri} Queue[X].coreC}^{\text{Insight 2}}$$

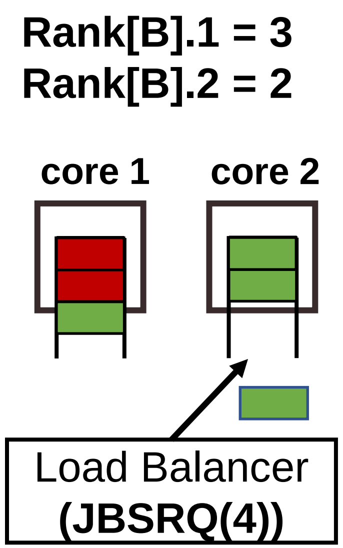
[Insight 1] rank is contributed by same/higher priority requests.

[Insight 2] rank is contributed less by lower priority requests.

JBSRQ Examples:

Service A's Prio = Hi
Service B's Prio = Lo
λ = 0.2

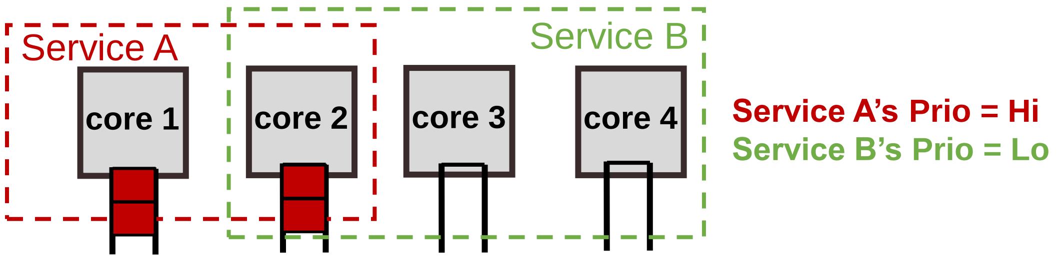


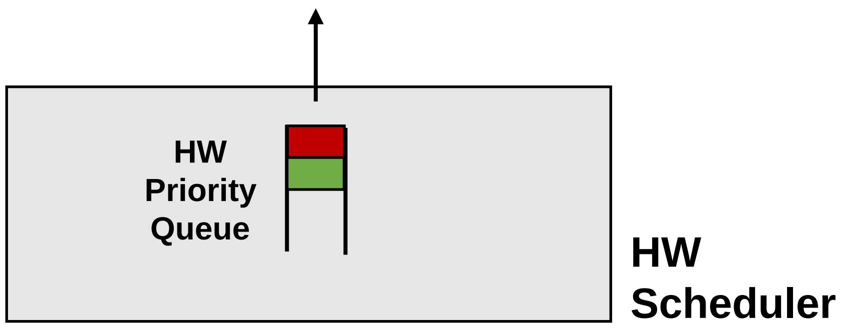


JBSRQ cooperates with the host priority queue and achieves optimal for both Hi/Lo priority requests!

Q3: Architecture of the on-NIC load balancer and scheduler

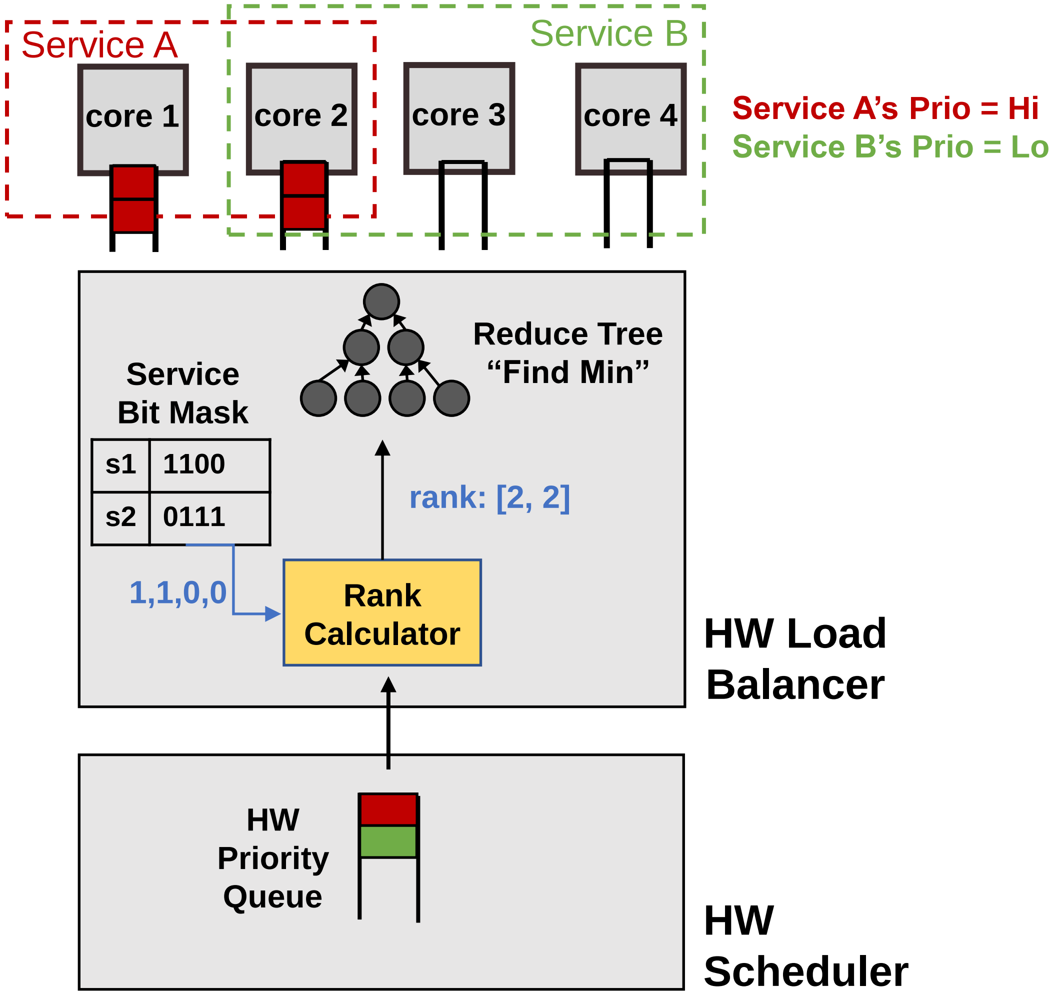
- Hardware request scheduler :
    - A hardware priority queue sorts services and dequeues the frontmost service.





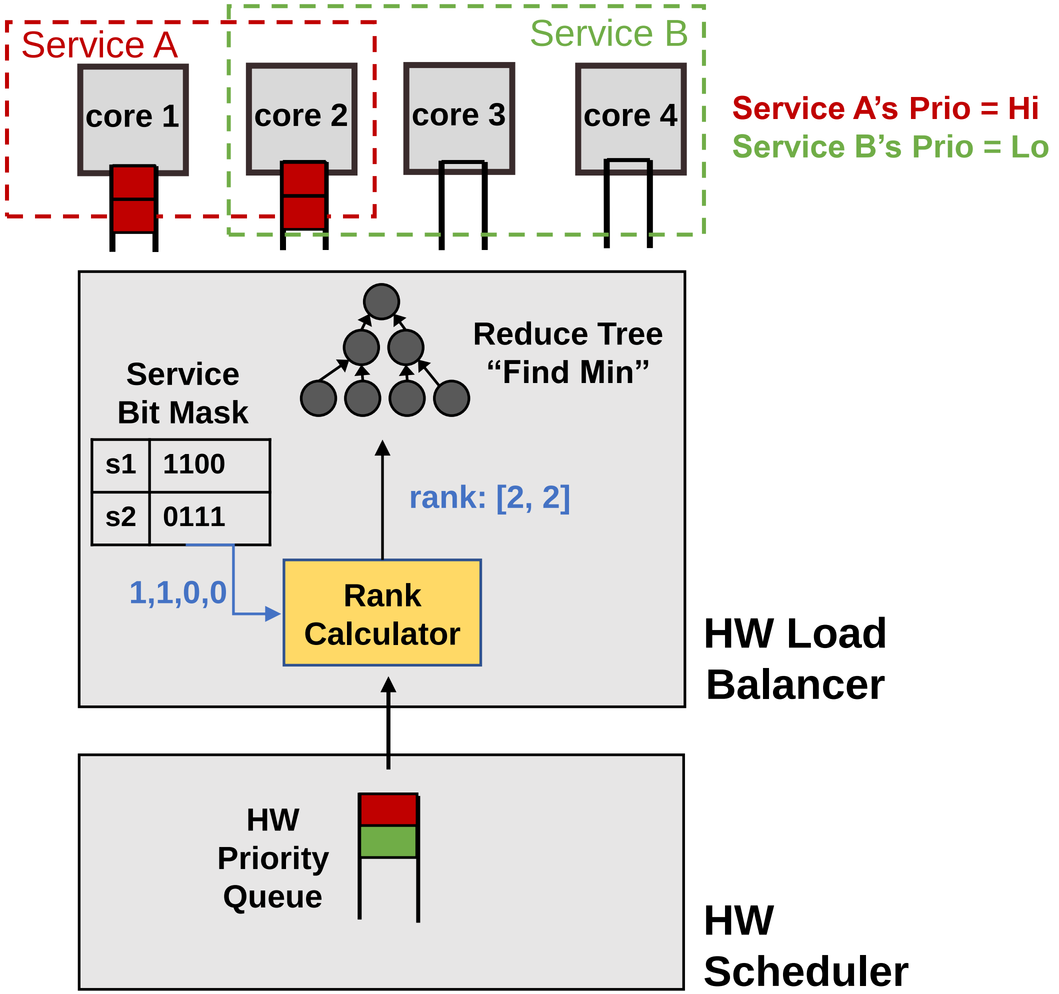
Q3: Architecture of the on-NIC load balancer and scheduler

- Hardware request scheduler :
  - A hardware priority queue sorts services and dequeues the frontmost service.
- Hardware load balancer:
  - Find the service-to-core mapping.
  - Calculate rank.
  - Find the minimal ranked core.
  - **If that core's rank < N, dispatch this request to that core.**



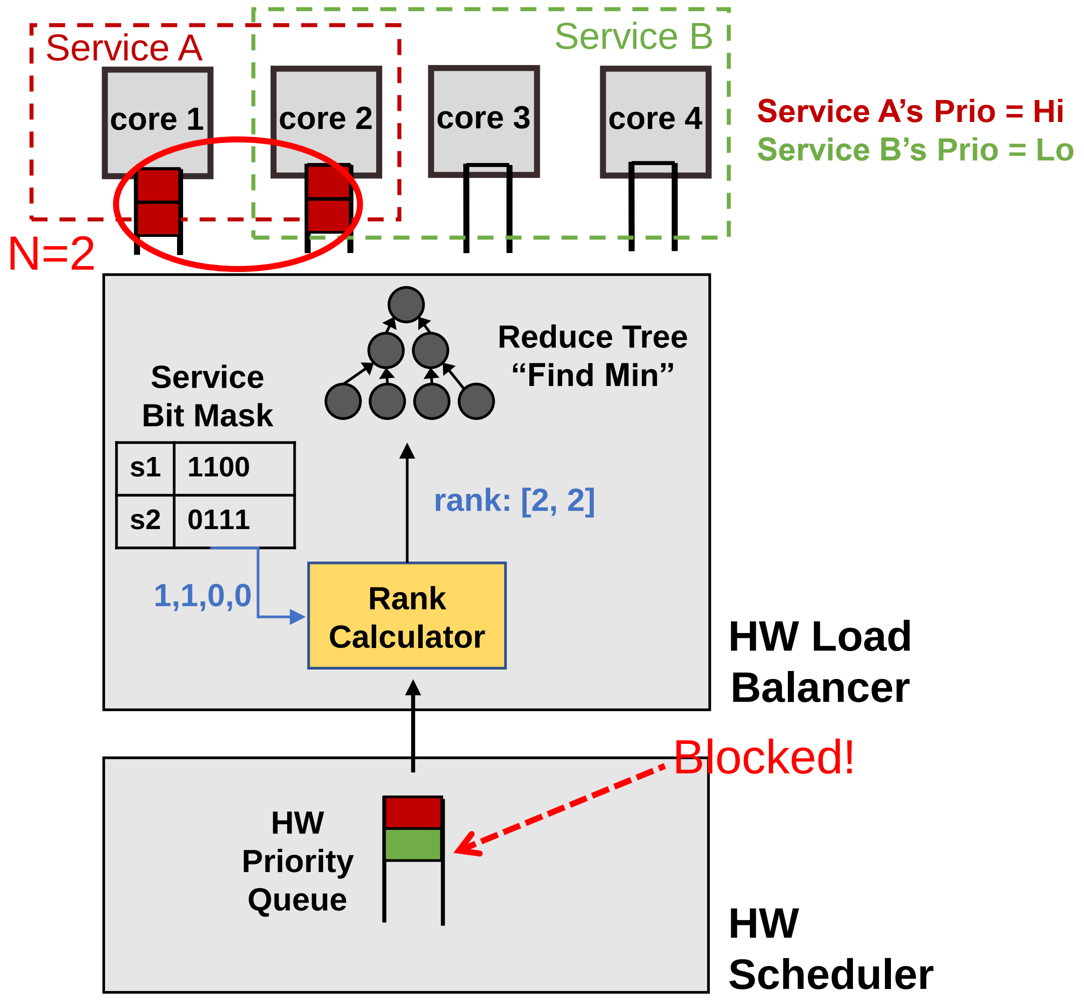
Q3: Architecture of the on-NIC load balancer and scheduler

- Hardware request scheduler :
  - A hardware priority queue sorts services and dequeues the frontmost service.
- Hardware load balancer:
  - Find the service-to-core mapping.
  - Calculate rank.
  - Find the minimal ranked core.
  - **If that core's rank < N, dispatch this request to that core.**
- **Problem:** the scheduler might schedule a request that cannot be dispatched by the JBSRQ.



Q3: Architecture of the on-NIC load balancer and scheduler

- Hardware request scheduler :
  - A hardware priority queue sorts services and dequeues the frontmost service.
- Hardware load balancer:
  - Find the service-to-core mapping.
  - Calculate rank.
  - Find the minimal ranked core.
  - **If that core's rank < N, dispatch this request to that core.**
- **Problem:** the scheduler might schedule a request that cannot be dispatched by the JBSRQ.

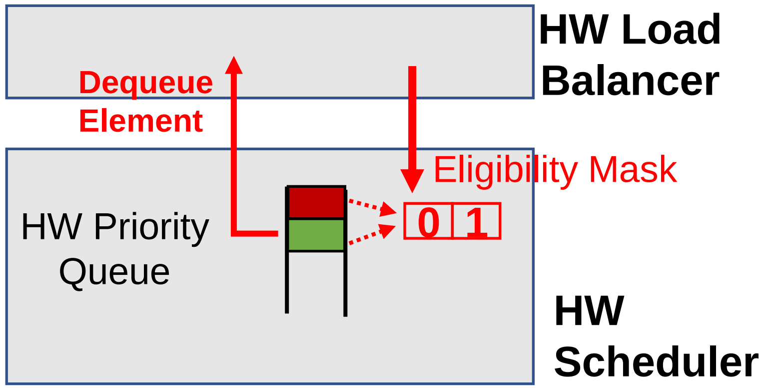


Non-blocking interface between the on-NIC load balancer and scheduler

Interface: Eligibility Mask

Eligibility of a service: cores running this service have at least one core with a rank smaller than the bound.

The hardware scheduler dequeues the **front-most eligible** element.



More details in our paper

- NIC-assisted CPU reallocation.
  - NIC generates reallocation hints at very fine granularity (e.g., every 5 us).
- Low overhead NIC-host metadata communication.
  - ~50M messages per second through MMIO.
  - Further decrease the overhead through adaptive inlining.

Implementation

- **100G FPGA prototype of the Ringleader NIC:** implemented in 4K lines of Verilog code. Run at100G, use a 250 MHz frequency.
- **User space NIC driver:** implemented in 1.5K lines of C code and provides a DPDK-like kernel-bypass access to the NIC.
- **Integrate with the Datapath OS:** we integrated our NIC driver with the Demikernel libOS using 800 lines of Rust.

Evaluation

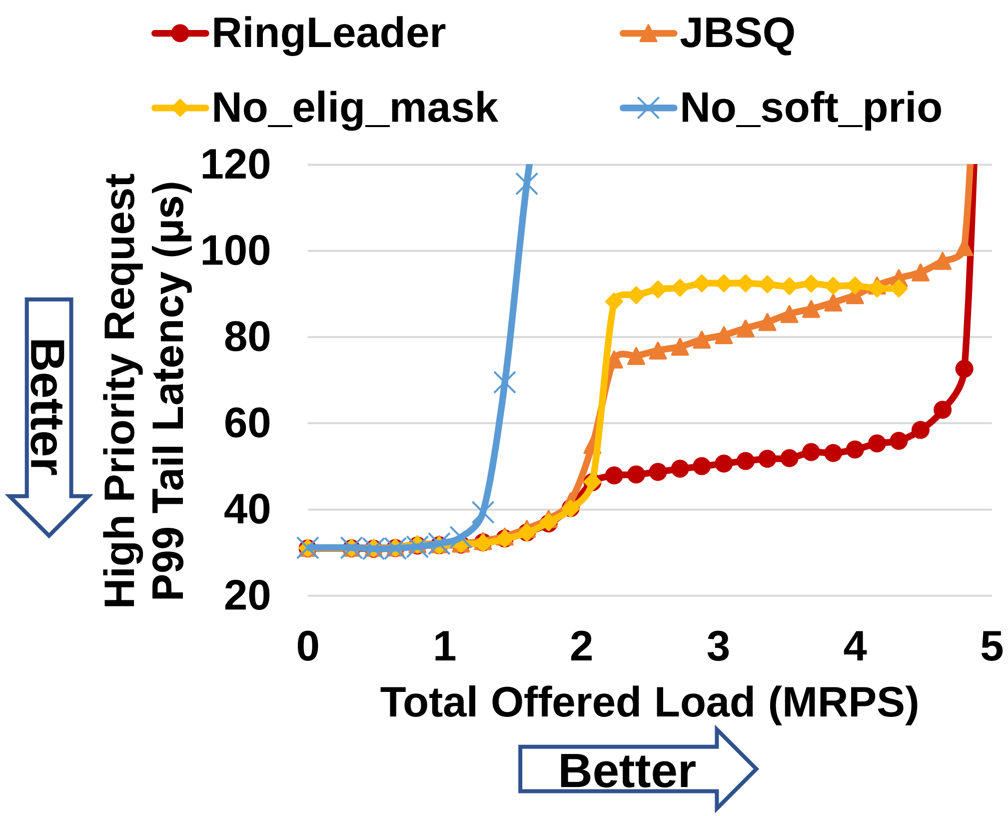
Workloads:

- Synthetic benchmark with different service time distributions.
- RocksDB in-memory database.

Baselines:

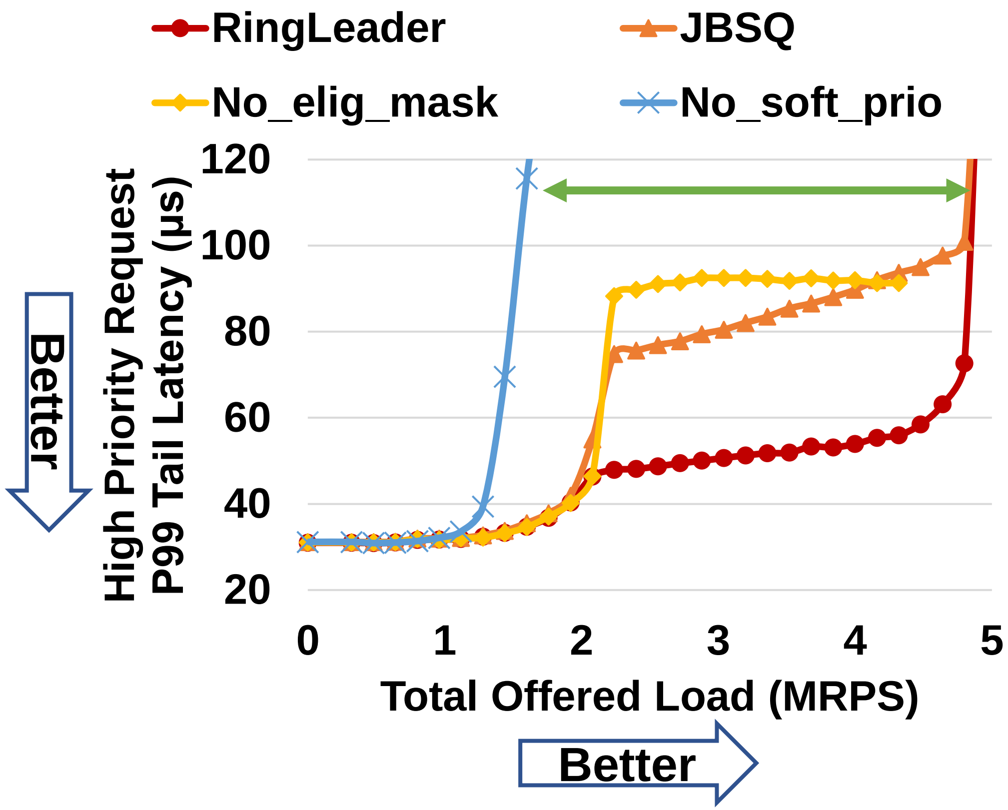
- **Shinjuku (NSDI'19):** software-based centralized request load balancing and scheduling.
- **Caladan (OSDI'20):** software-based fast CPU reallocation.
- **RSS:** NIC RSS to spread requests to cores using random hash.

Q1: How Ringleader's design decision contributes to its overall performance



Workloads	Description
High Bimodal (99-3,1-100)	99% requests are high priority, take 3 μsec. 1% requests are low priority, take 100 μsec.

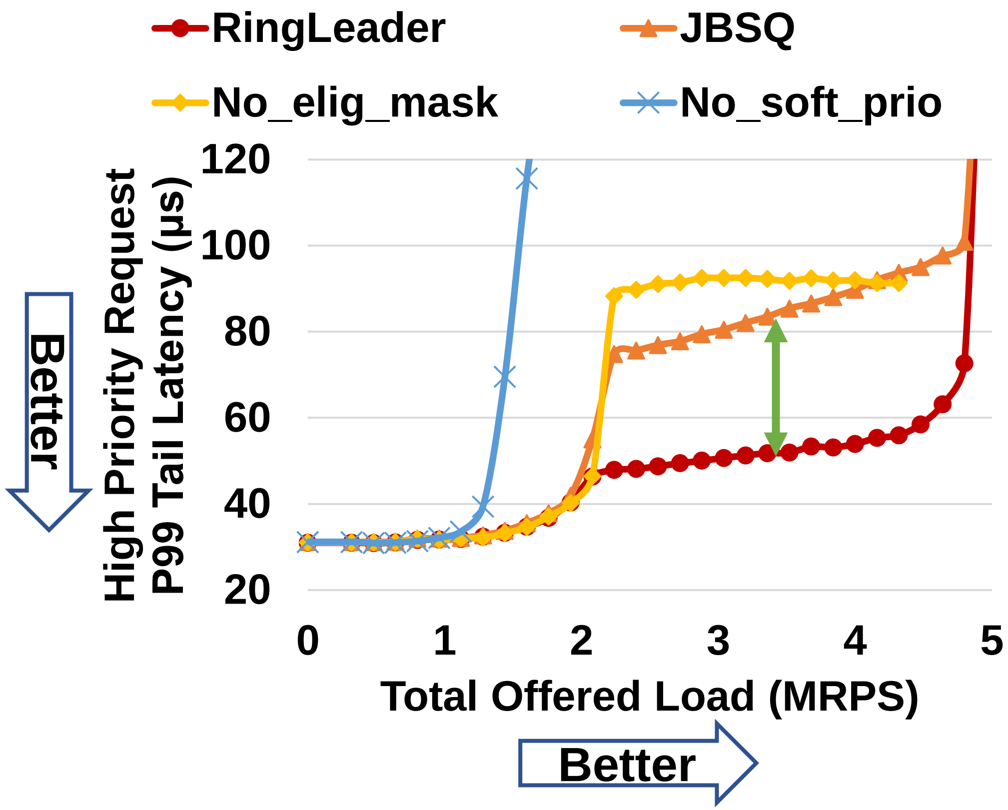
Q1: How Ringleader's design decision contributes to its overall performance



Workloads	Description
High Bimodal (99-3,1-100)	99% requests are high priority, take 3 μsec. 1% requests are low priority, take 100 μsec.

Remove Software Priority Queue: HoL bocking insides the host buffer.

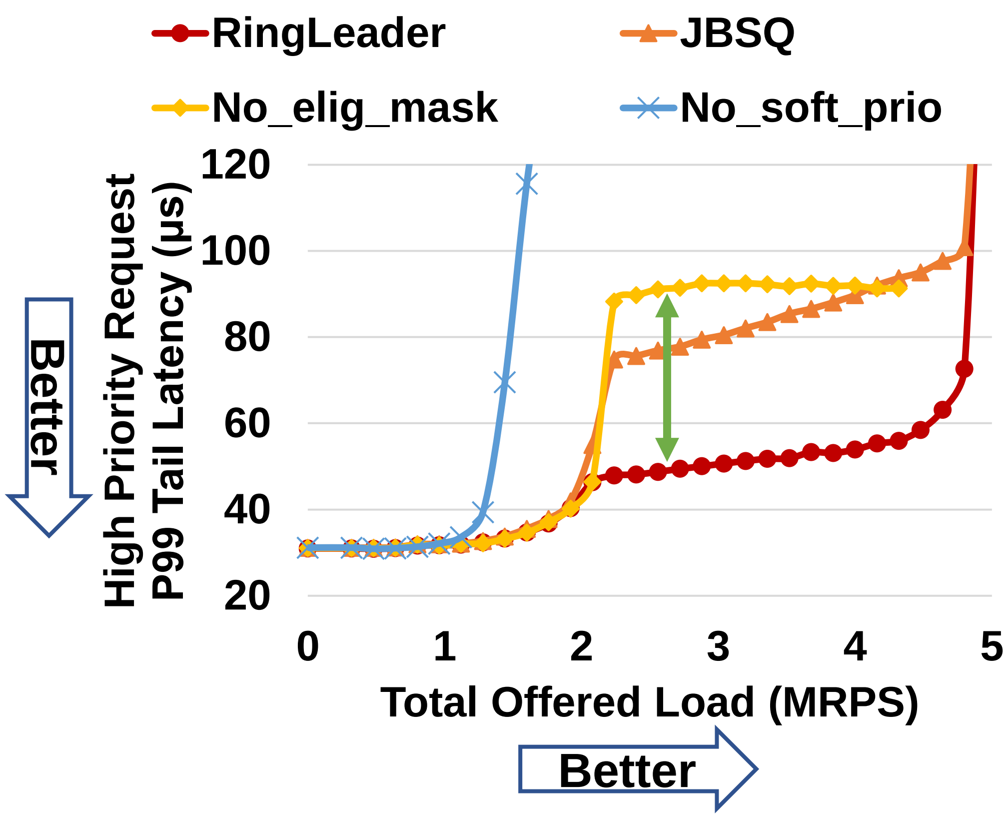
Q1: How Ringleader's design decision contributes to its overall performance



Workloads	Description
High Bimodal (99-3,1-100)	99% requests are high priority, take 3 μsec. 1% requests are low priority, take 100 μsec.

Disable JBSRQ: Suboptimal dispatching policy.

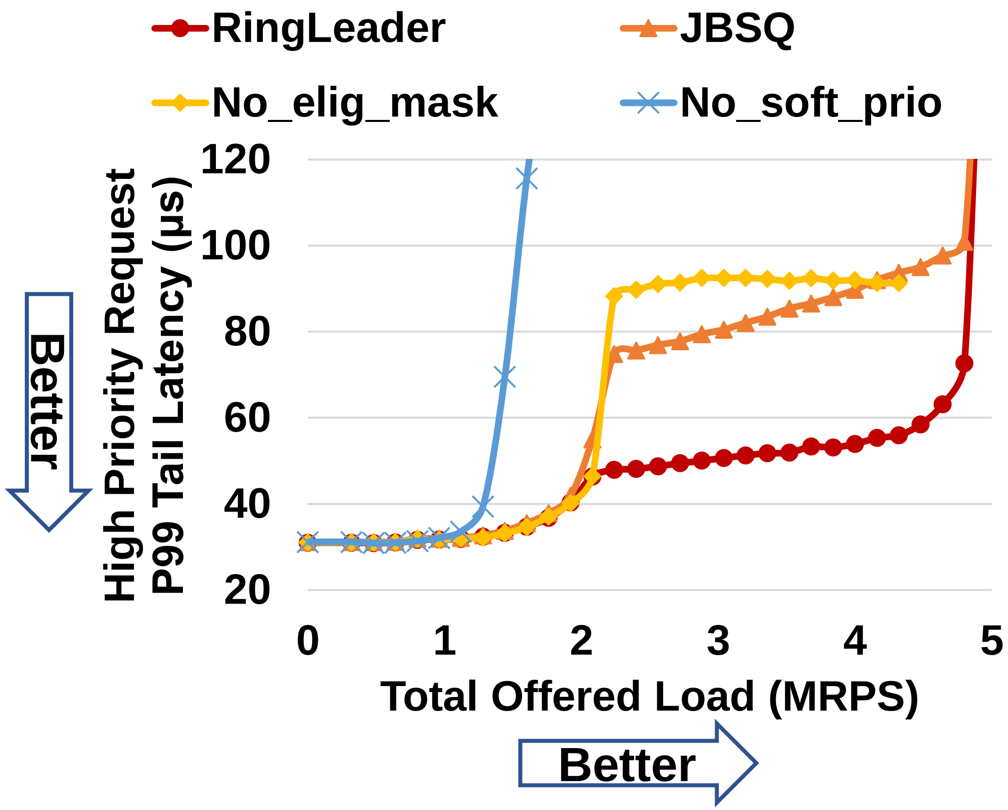
Q1: How Ringleader's design decision contributes to its overall performance



Workloads	Description
High Bimodal (99-3,1-100)	99% requests are high priority, take 3 μsec. 1% requests are low priority, take 100 μsec.

Disable eligibility mask: Hardware pipeline blocking.

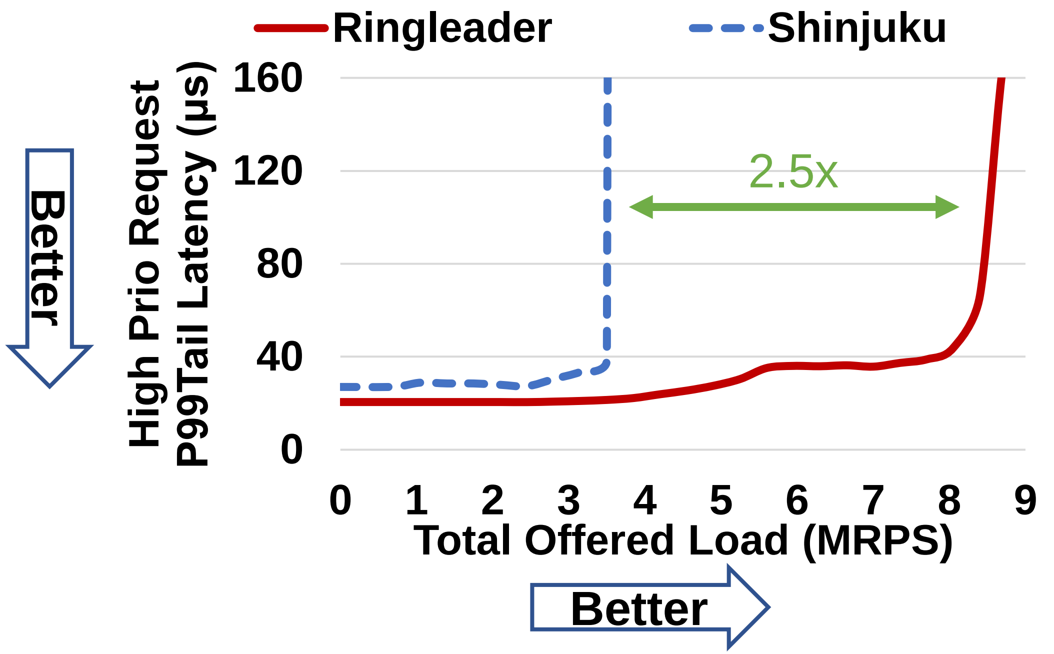
Q1: How Ringleader's design decision contributes to its overall performance

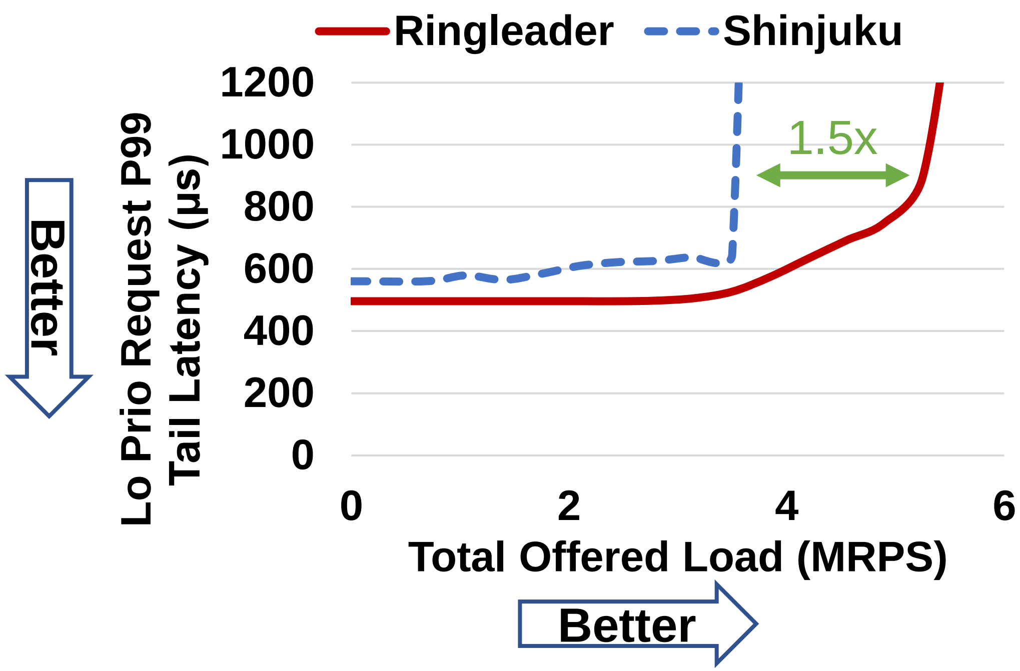


Workloads	Description
High Bimodal (99-3,1-100)	99% requests are high priority, take 3 μsec. 1% requests are low priority, take 100 μsec.

Takeaways: Software priority queues, JBSRQ, and the eligibility mask ensure that Ringleader can achieve effective orchestration.

Q2: How does Ringleader compared to the CPU-based orchestration





Takeaways: Ringleader achieves better performance and scalability than the software-based approach!

Conclusion

- RingLeader offloads orchestration through a new load balancing algorithm and scheduler, as well as a new OS/NIC interface.

- Experiments on a 100 Gbps FPGA NIC show that RingLeader offers good tail latency and high throughput.

https://github.com/utnslab/RingleaderNIC