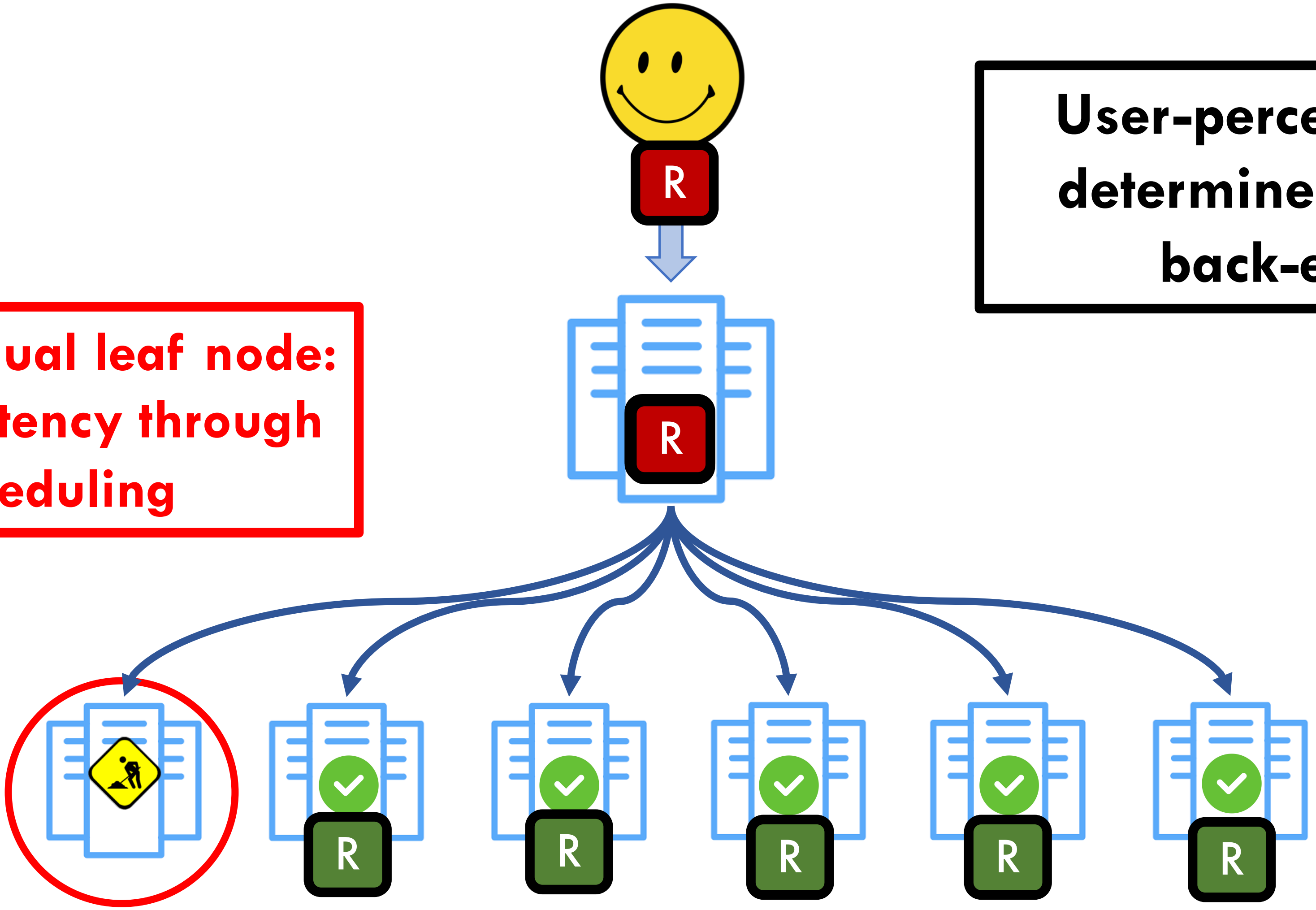
Shinjuku: Preemptive Scheduling for Microsecond-Scale Tail Latency

Kostis Kaffes, Timothy Chong, Jack Tigar Humphries,
Adam Belay, David Mazières, Christos Kozyrakis

Tail latency matters for datacenter workloads

User-perceived latency determined by slowest back-end node

Focus on individual leaf node: minimize tail latency through better scheduling



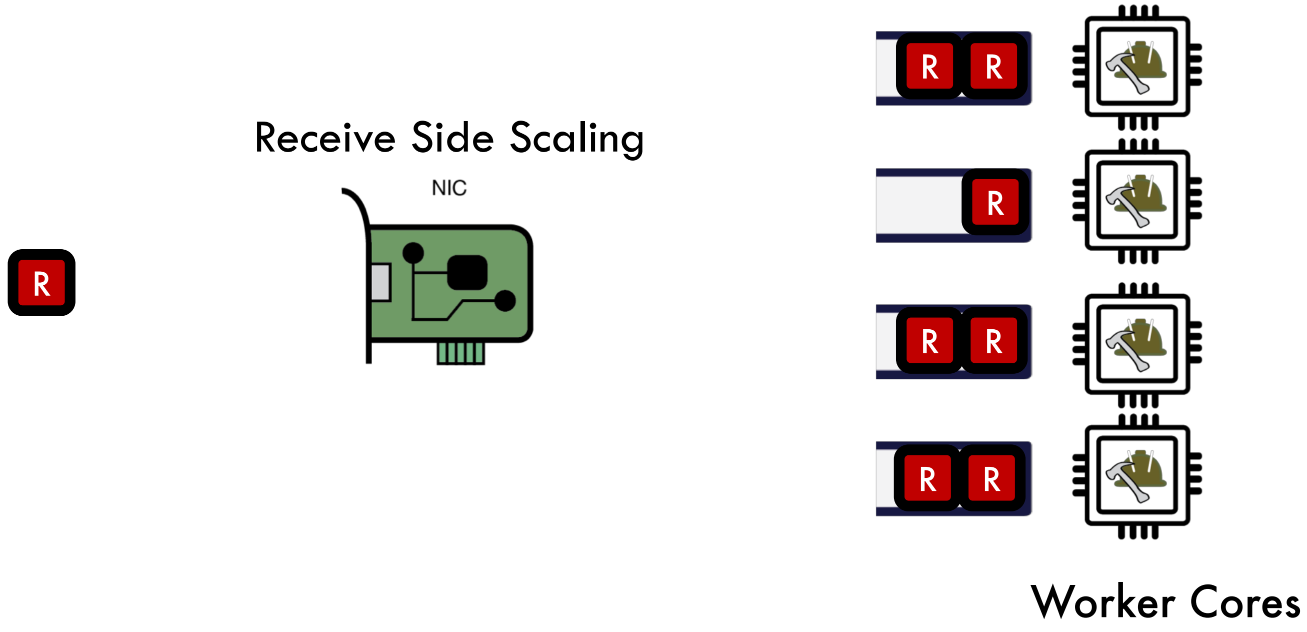
Achieving low tail latency at microsecond scale is hard

Problem: High OS overheads

Solution: OS Bypass, polling (no interrupts), run-to-completion (no scheduling)

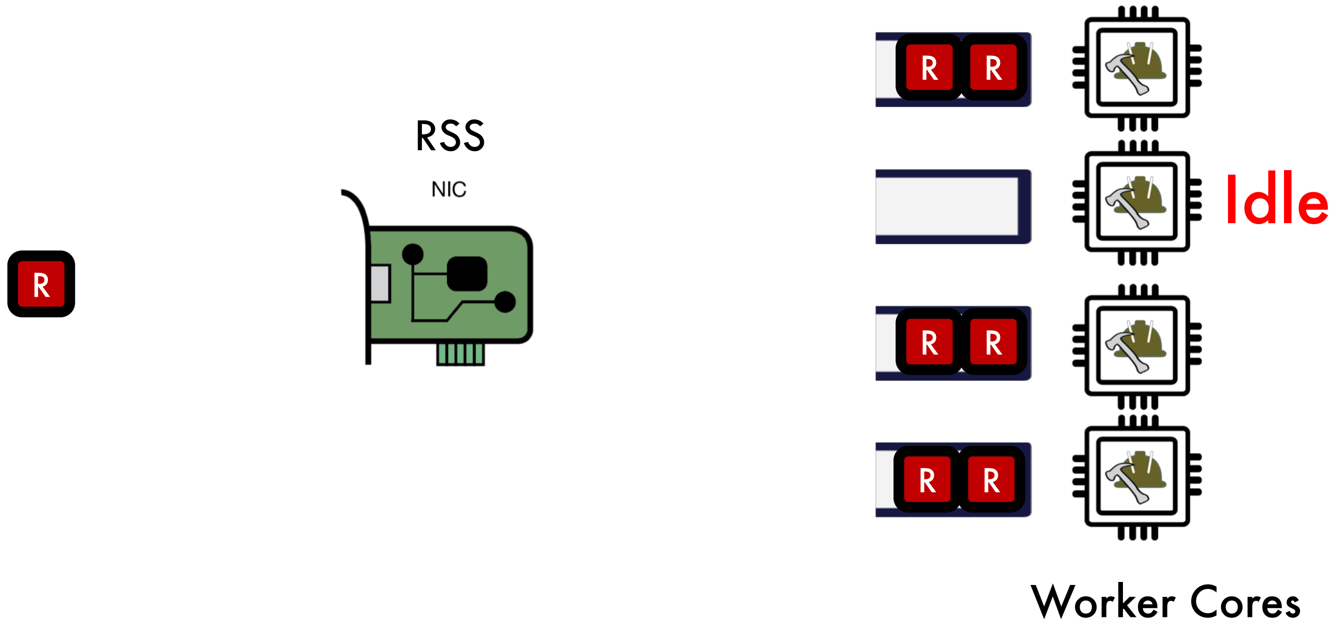
Distributed Queues + First Come First Serve scheduling

d-FCFS (DPDK, IX, Arrakis)



Achieving low tail latency at microsecond scale is hard

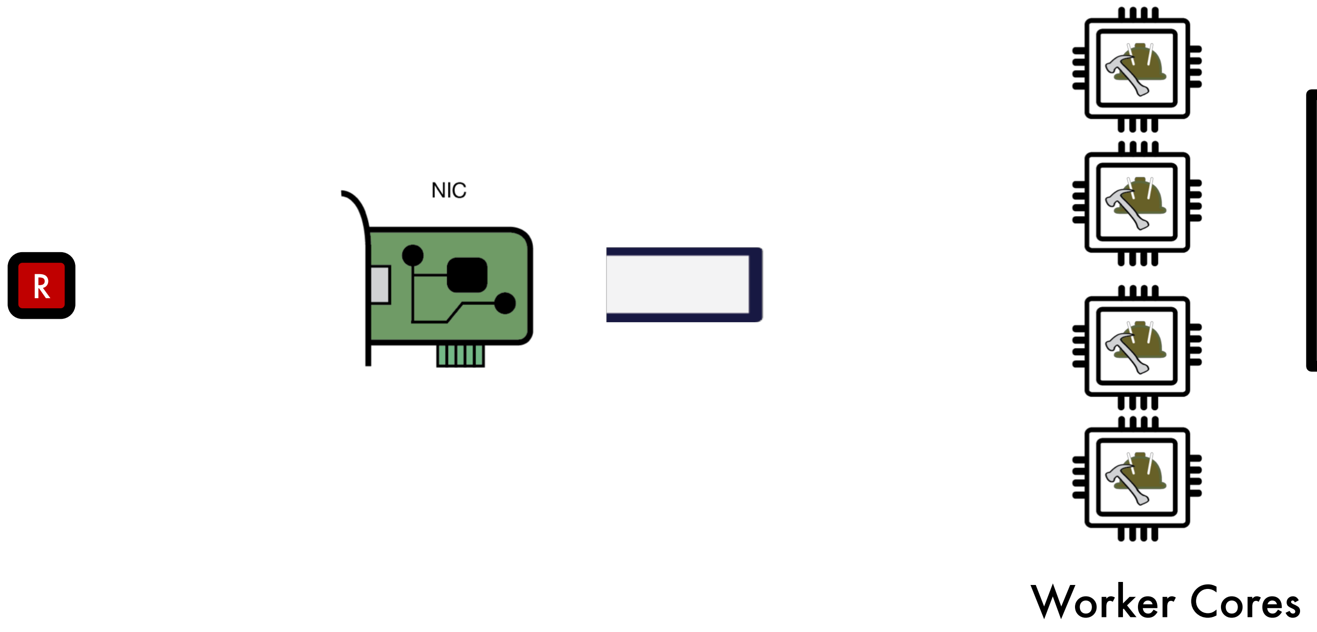
Problem: Queue imbalance because d-FCFS is not work conserving



Achieving low tail latency at microsecond scale is hard

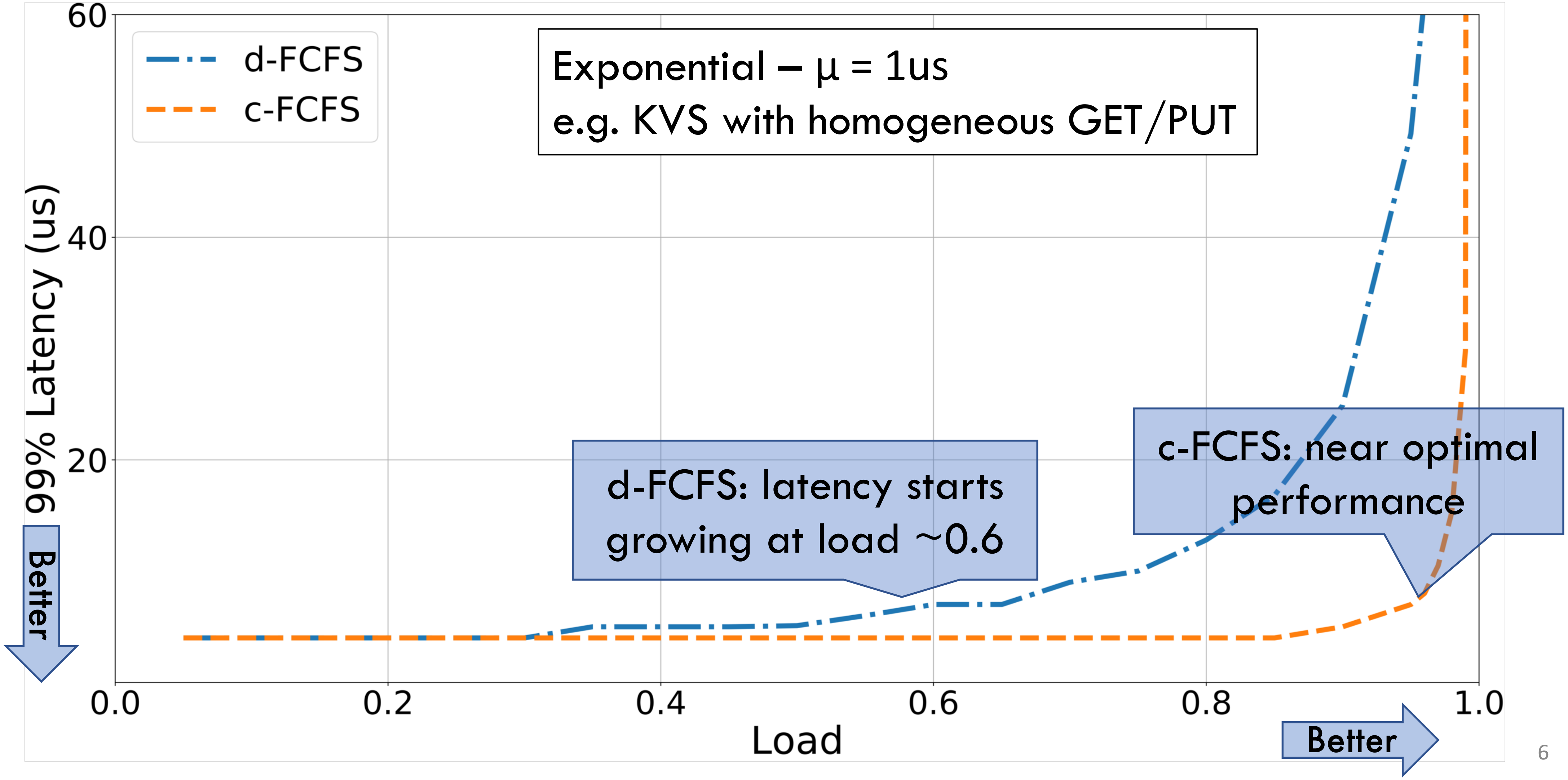
Problem: Queue imbalance because d-FCFS is not work conserving

Solution: Centralized queue - **c-FCFS**

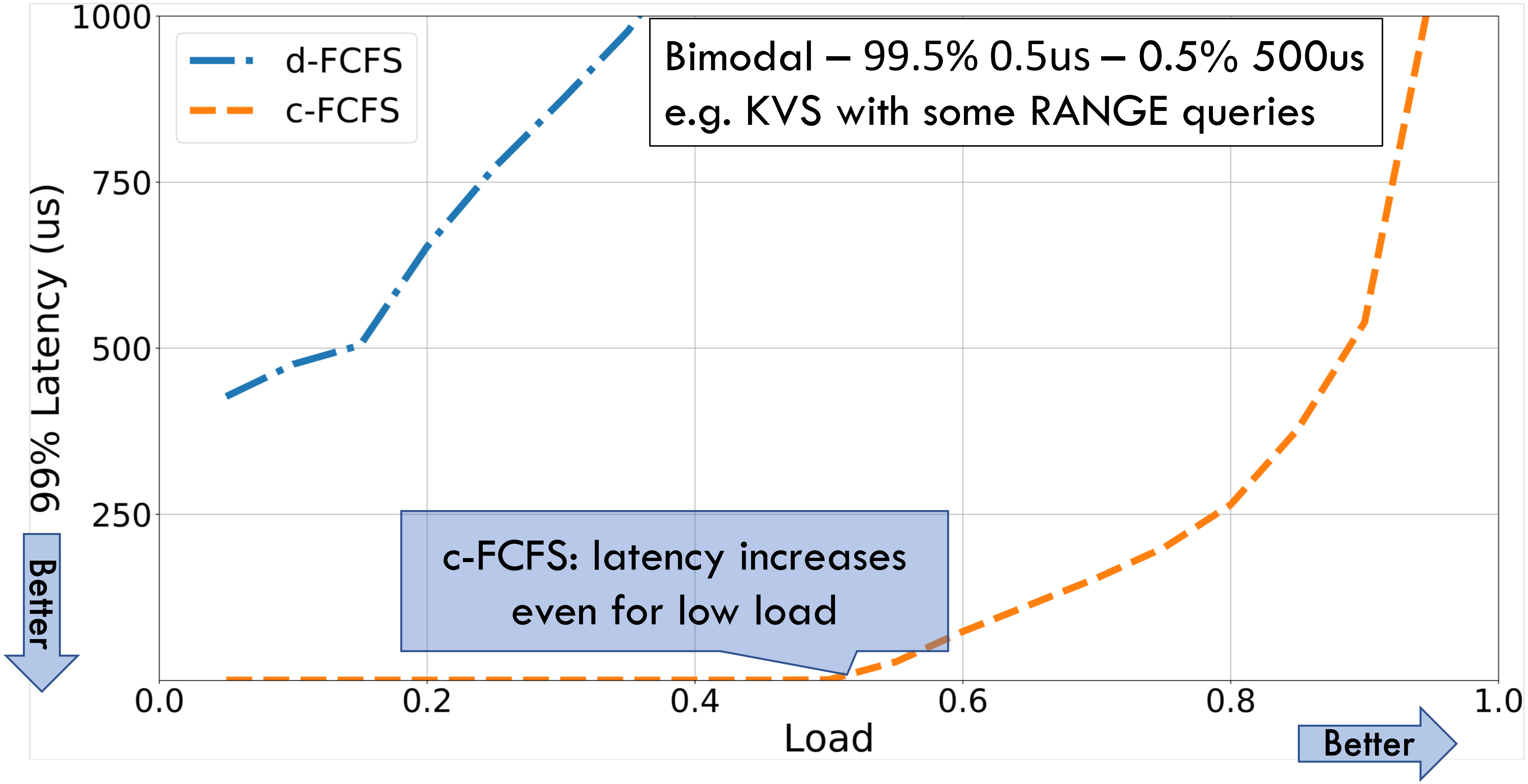


Approximation:
d-FCFS + stealing
e.g., ZygOS

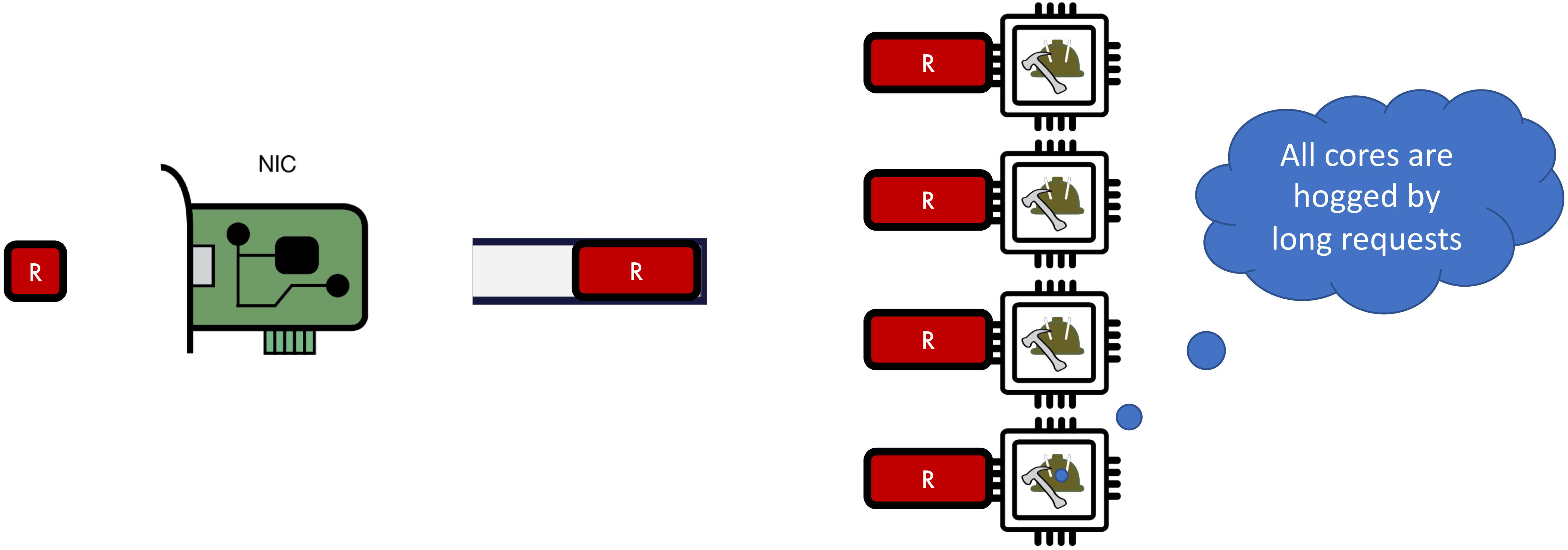
Ideal centralized queue is better in simulation



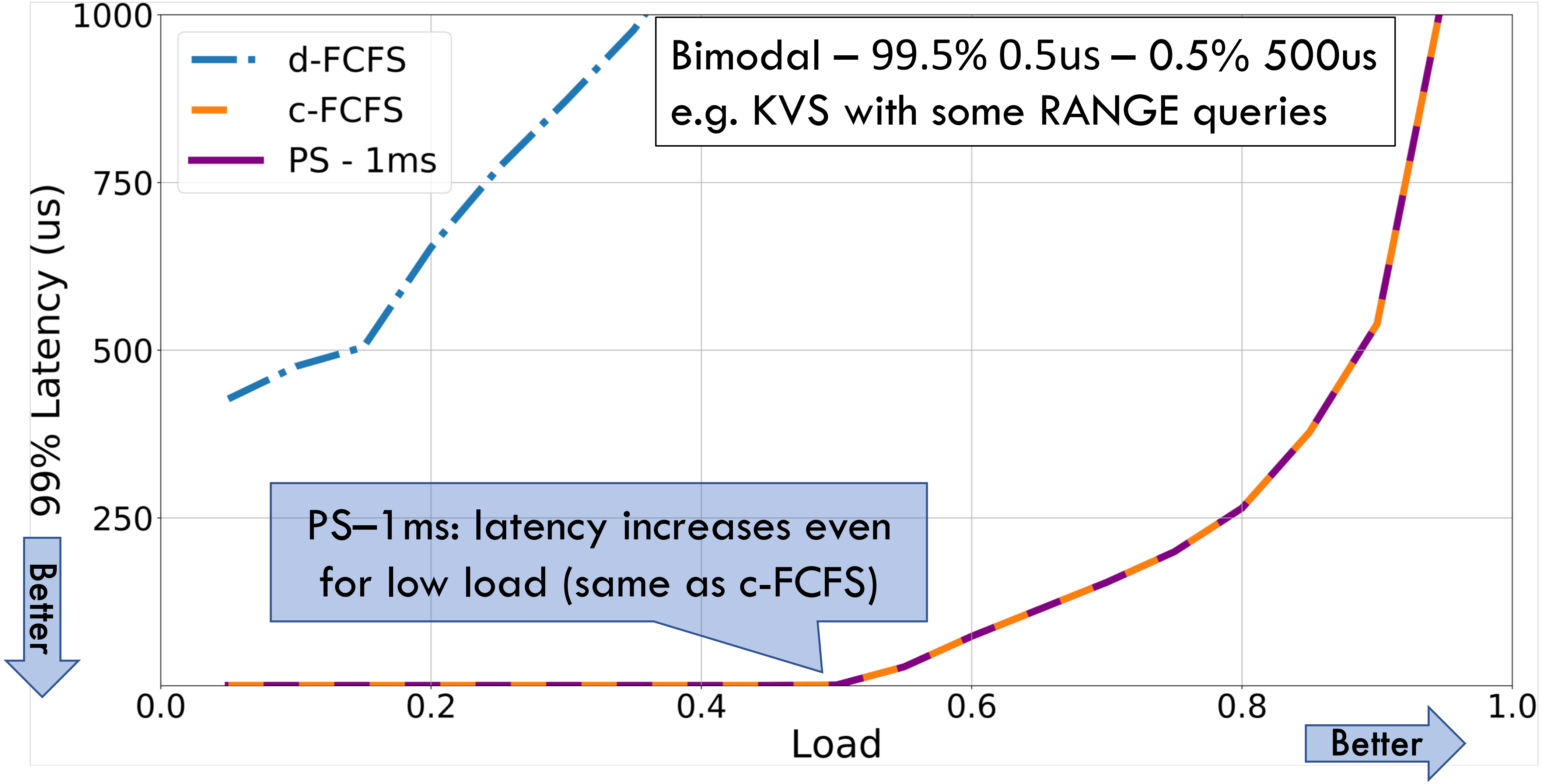
Is FCFS good enough when task duration varies?



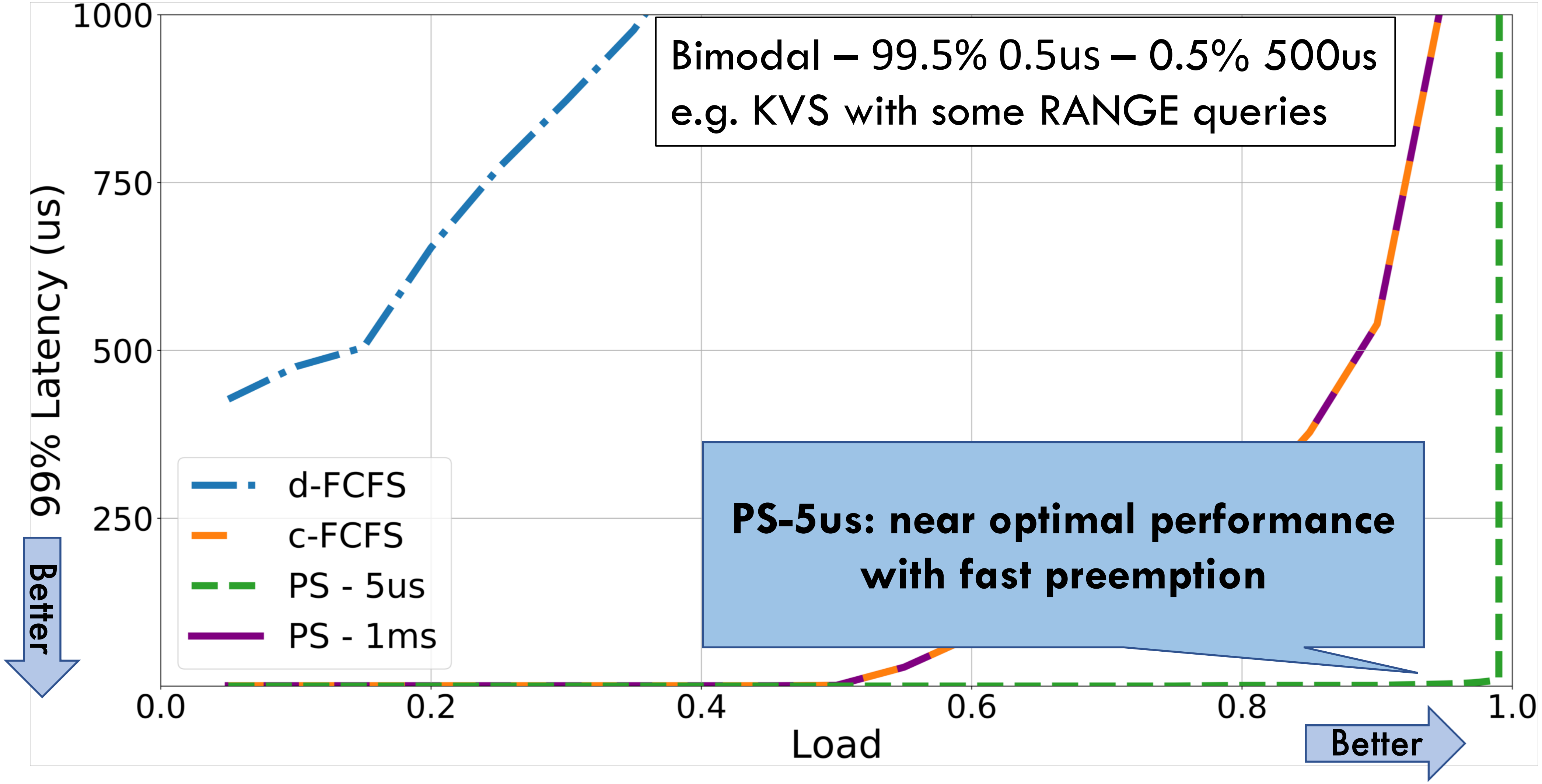
Problem: Short requests get stuck behind long ones



What if we could use the same preemptive scheduling as Linux?



Solution: What if we could use preemptive scheduling but at usec scale?



Insights

Effective scheduling for tail latency requires:

- Centralized queue
- Preemption
- Scheduling policies tailored for each workload

Problem: Microsecond scale requires

- Millions of queue accesses per second
- Preemption as often as every 5us
- Light-weight scheduling policies

Solution: Shinjuku

A single address-space operating system that achieves microsecond-scale tail latency for all types of workloads regardless of variability in task duration

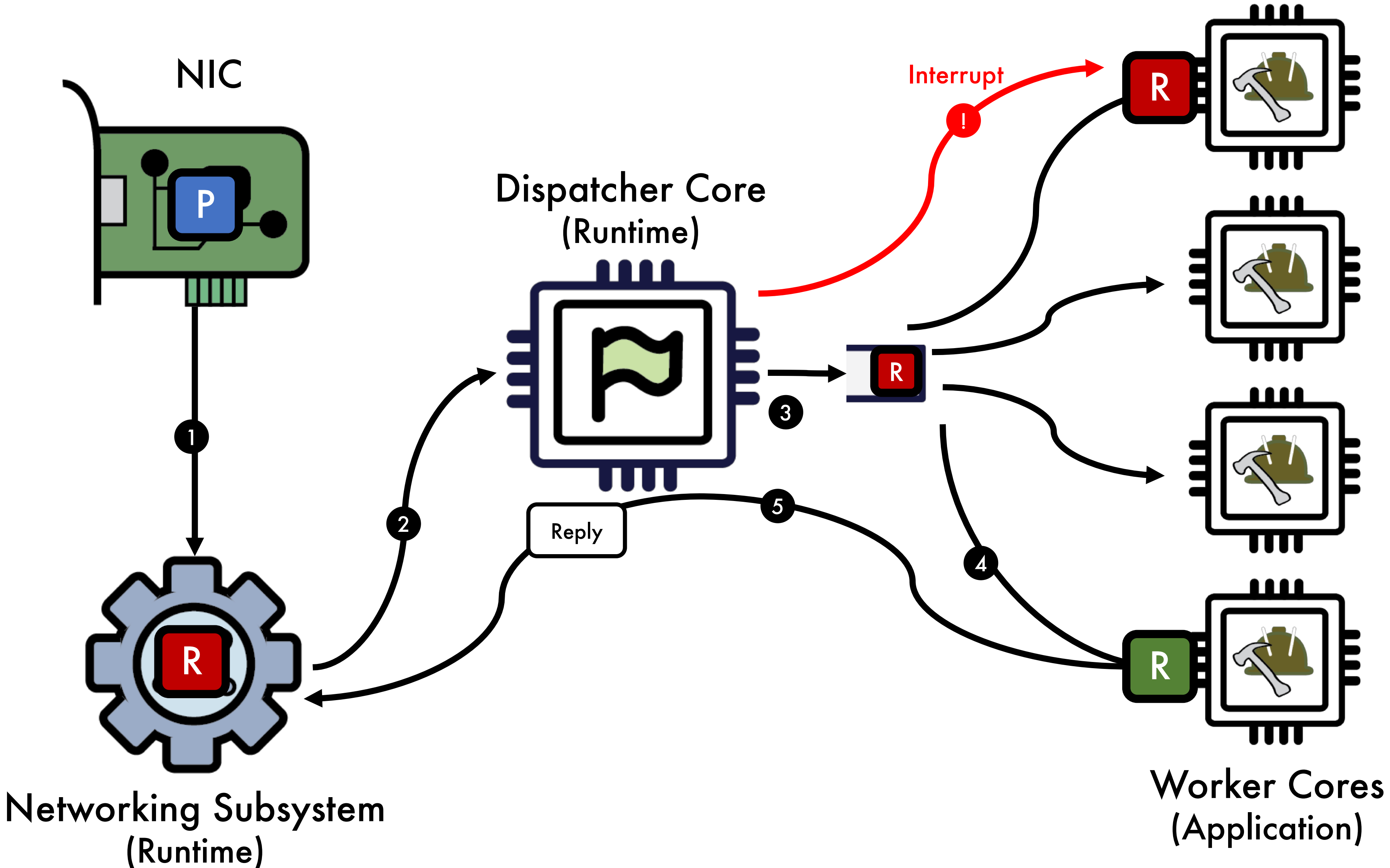
Key Features:

- Dedicated core for scheduling and queue management
- Leverage hardware support for virtualization for fast preemption
- Very fast context switching in user space
- Match scheduling policy to task distribution and target latency

Outline

- Shinjuku Design
- Preemption Mechanisms
- Scheduling Policies
- Evaluation

Shinjuku Design



1. Process packets and generate application-level requests
2. Pass requests to centralized dispatcher using shared memory
3. Add requests to centralized queue
4. Schedule requests to worker cores using shared memory
5. Send replies back to clients through the networking subsystem

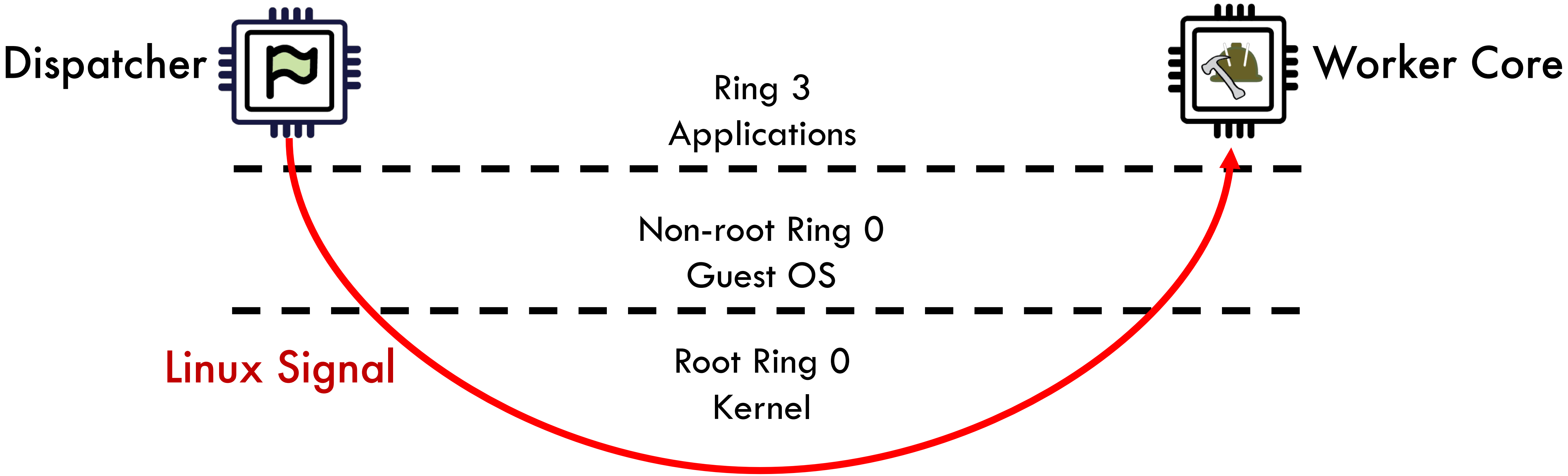
! Interrupt long running requests and schedule other requests from the queue

Outline

- Shinjuku Design
- Preemption Mechanisms
- Scheduling Policies
- Evaluation

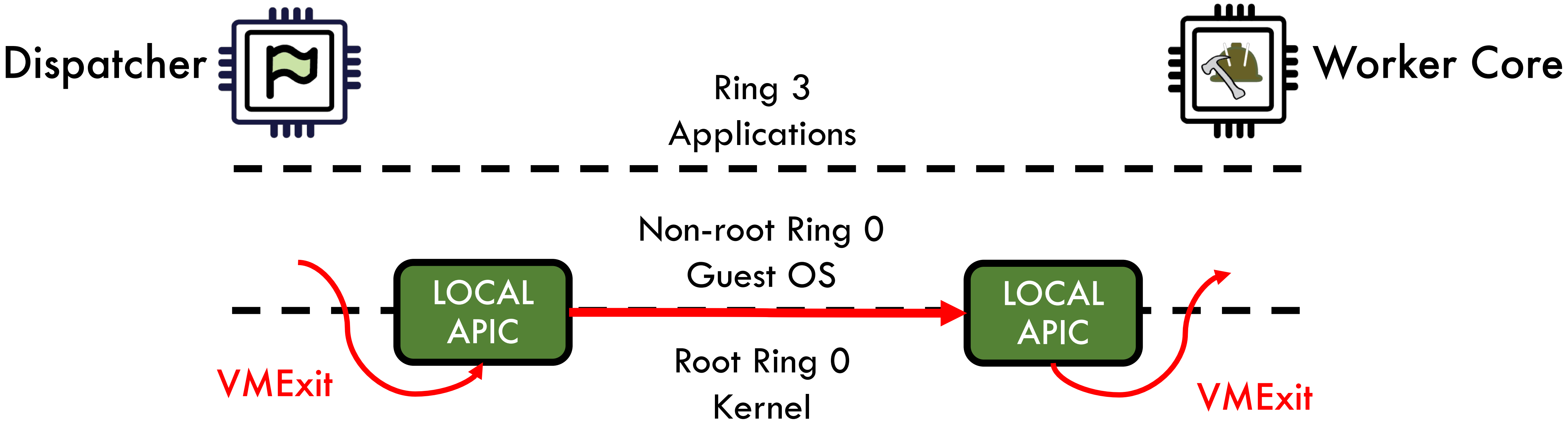
Minimizing Preemption Overhead

	Sender Overhead	Receiver Overhead
Linux Signal	2084 cycles	2523 cycles



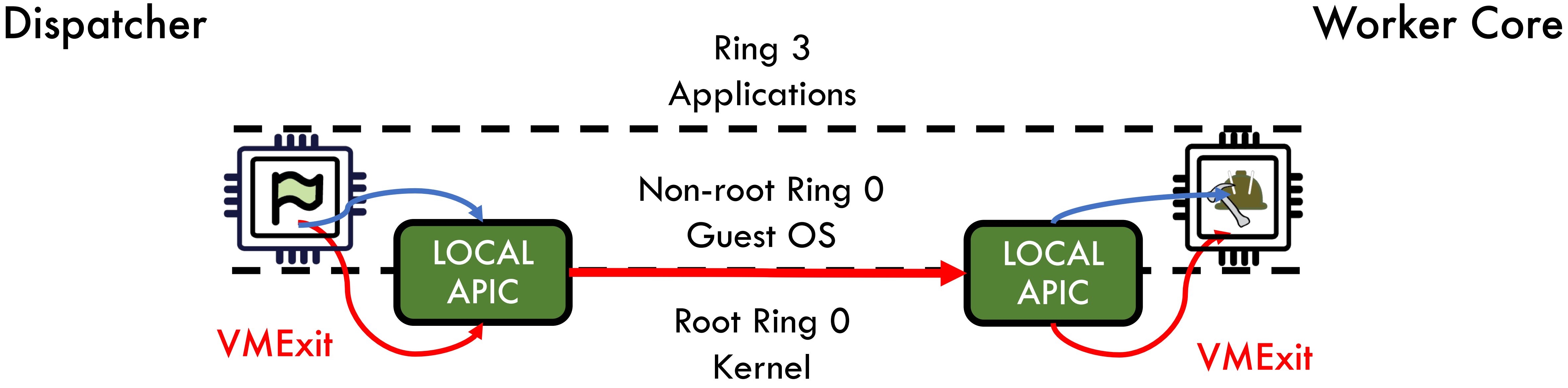
Minimizing Preemption Overhead

	Sender Overhead	Receiver Overhead
Linux Signal	2084 cycles	2523 cycles
Hardware Interrupts	2081 cycles	2662 cycles



Minimizing Preemption Overhead

	Sender Overhead		Receiver Overhead	
Linux Signal	2084 cycles	-85%	2523 cycles	-52%
Hardware Interrupts	2081 cycles		2662 cycles	
no VMExits	298 cycles		1212 cycles	



Map APIC to dispatcher's address space

Posted Interrupts

Minimizing Preemption Overhead

	Sender Overhead		Receiver Overhead	
Linux Signal	2084 cycles		2523 cycles	
Hardware Interrupts	2081 cycles	-85%	2662 cycles	-52%
no VMExits	298 cycles		1212 cycles	

Dispatcher

Worker Core

5us Time Slice

Map APIC to dispatcher's address space

Posted Interrupts

Outline

- Shinjuku Design
- Preemption Mechanisms
- Scheduling Policies
- Evaluation

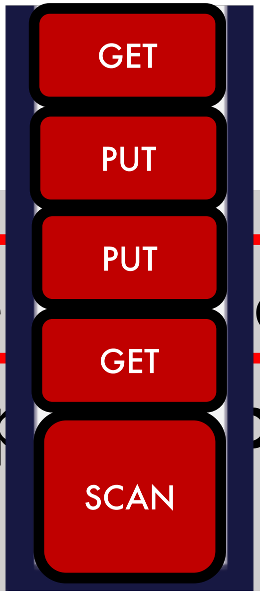
Scheduling policy

Case 1

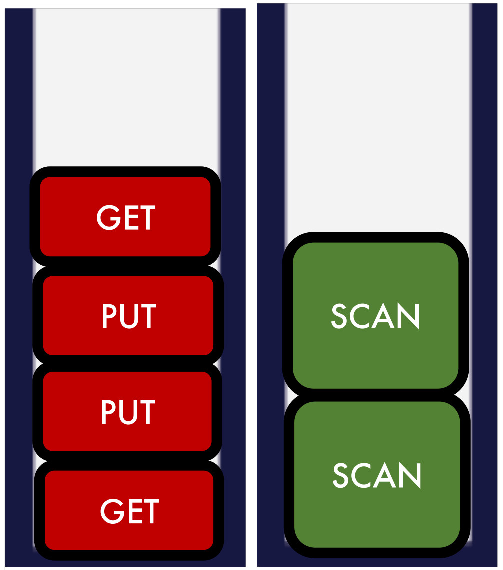
Case 2

1) Which que elect from?

2) Where to p preempted requests?

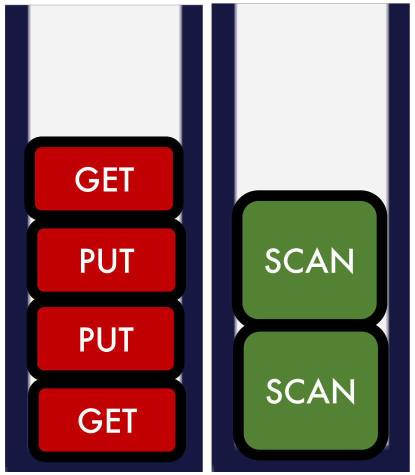


Single Queue (SQ)



Multiple Queues (MQ)

Queue Selection Policy



Multiple Queues (MQ)

<u>Policy:</u> Select the queue with the highest ratio: $\frac{Waiting\ Time}{Target\ Latency}$

<u>Short requests:</u> Initially low Target Latency → High Ratio

<u>Long requests:</u> Eventually high Waiting Time → High Ratio

Outline

- Shinjuku Design
- Preemption Mechanisms
- Scheduling Policies
- Evaluation

Evaluation

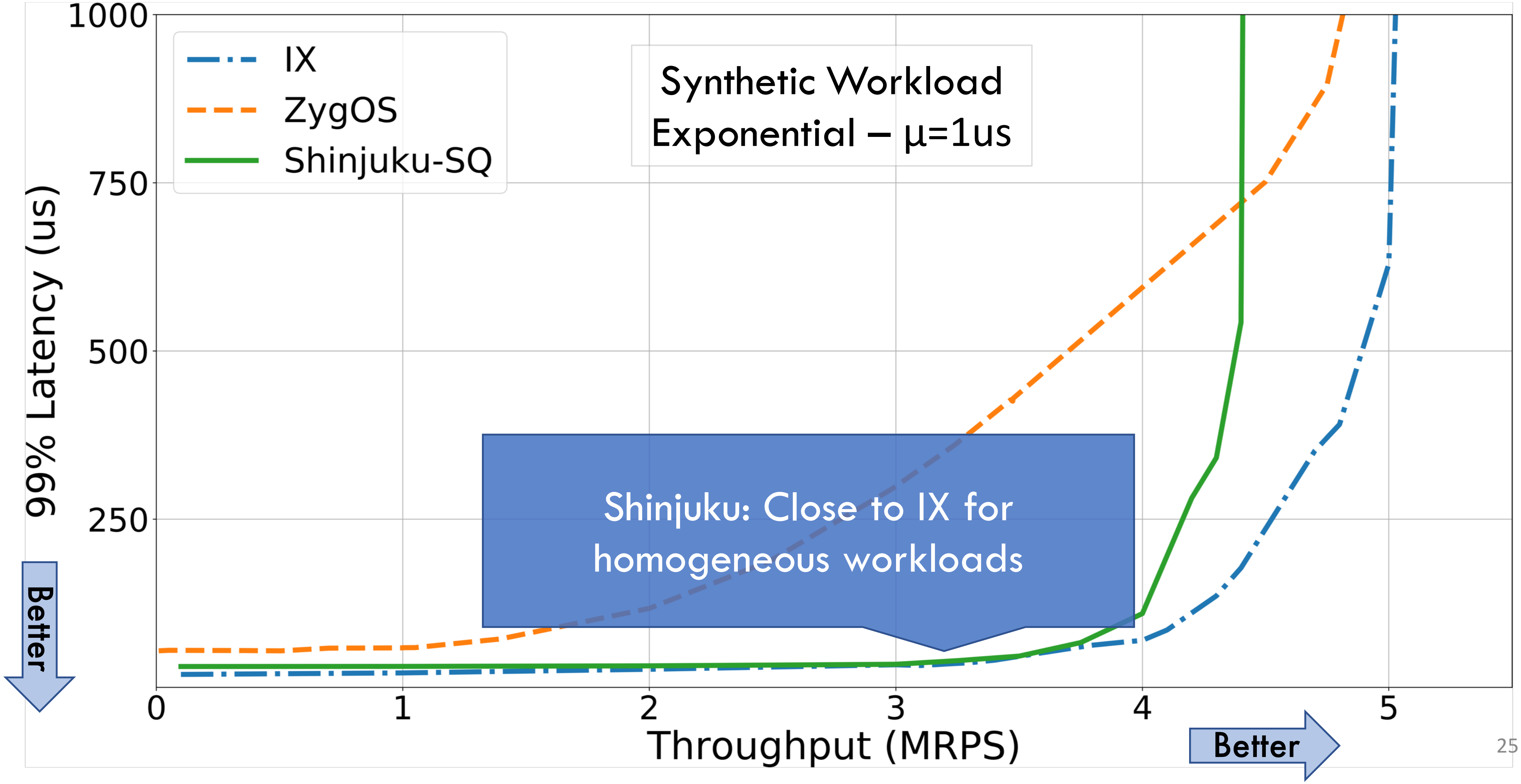
Systems

- ***Shinjuku*** – Centralized preemptive scheduling
  - 14 Logical Cores for workers
  - 1 Physical Core for both networker and dispatcher (1 Logical Core each)
- ***IX*** – d-FCFS
- ***ZygOS*** – d-FCFS + work stealing
  - 16 Logical Cores for workers

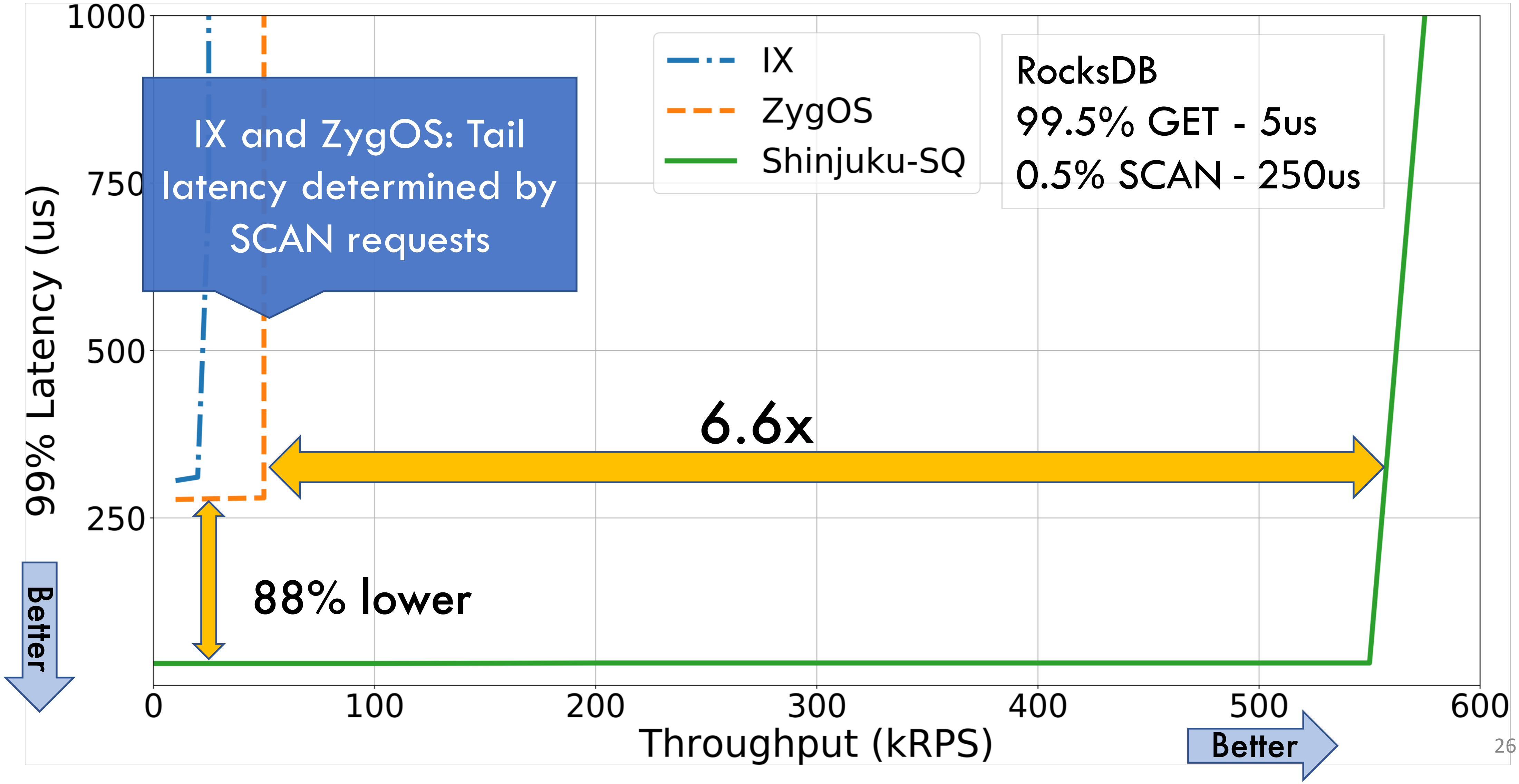
Workloads

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

Shinjuku under low variability

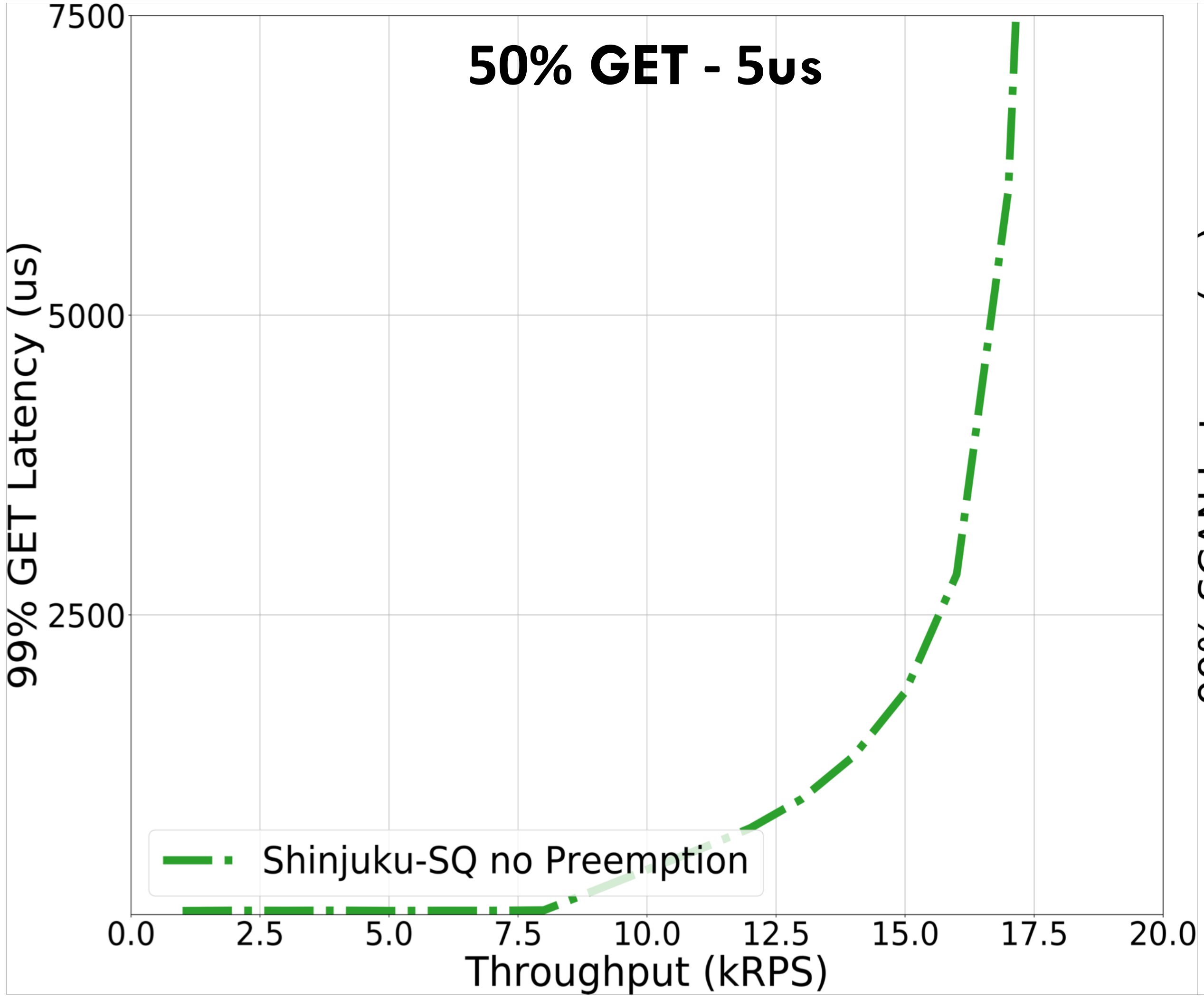


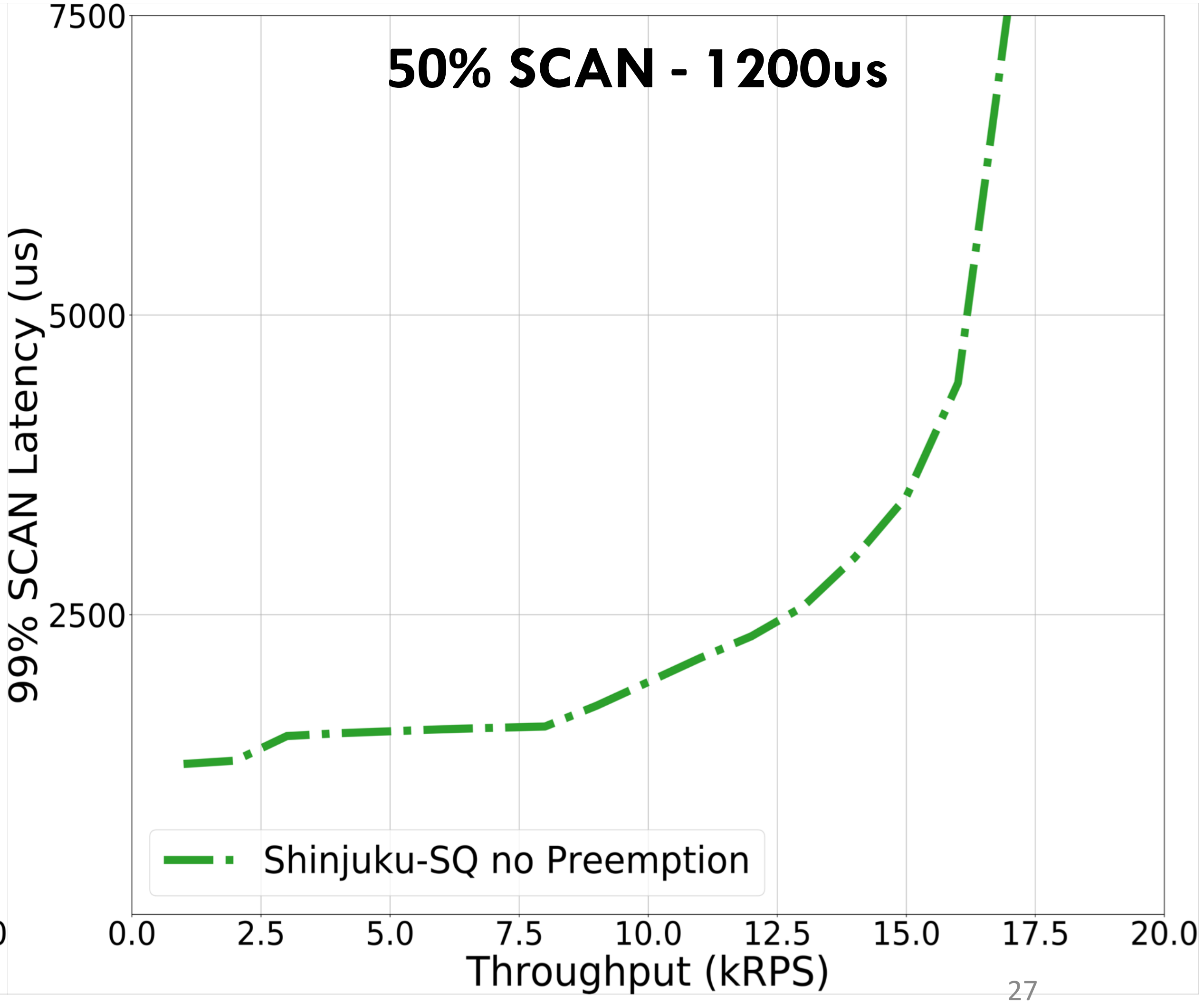
Shinjuku under high variability



How important is each optimization?

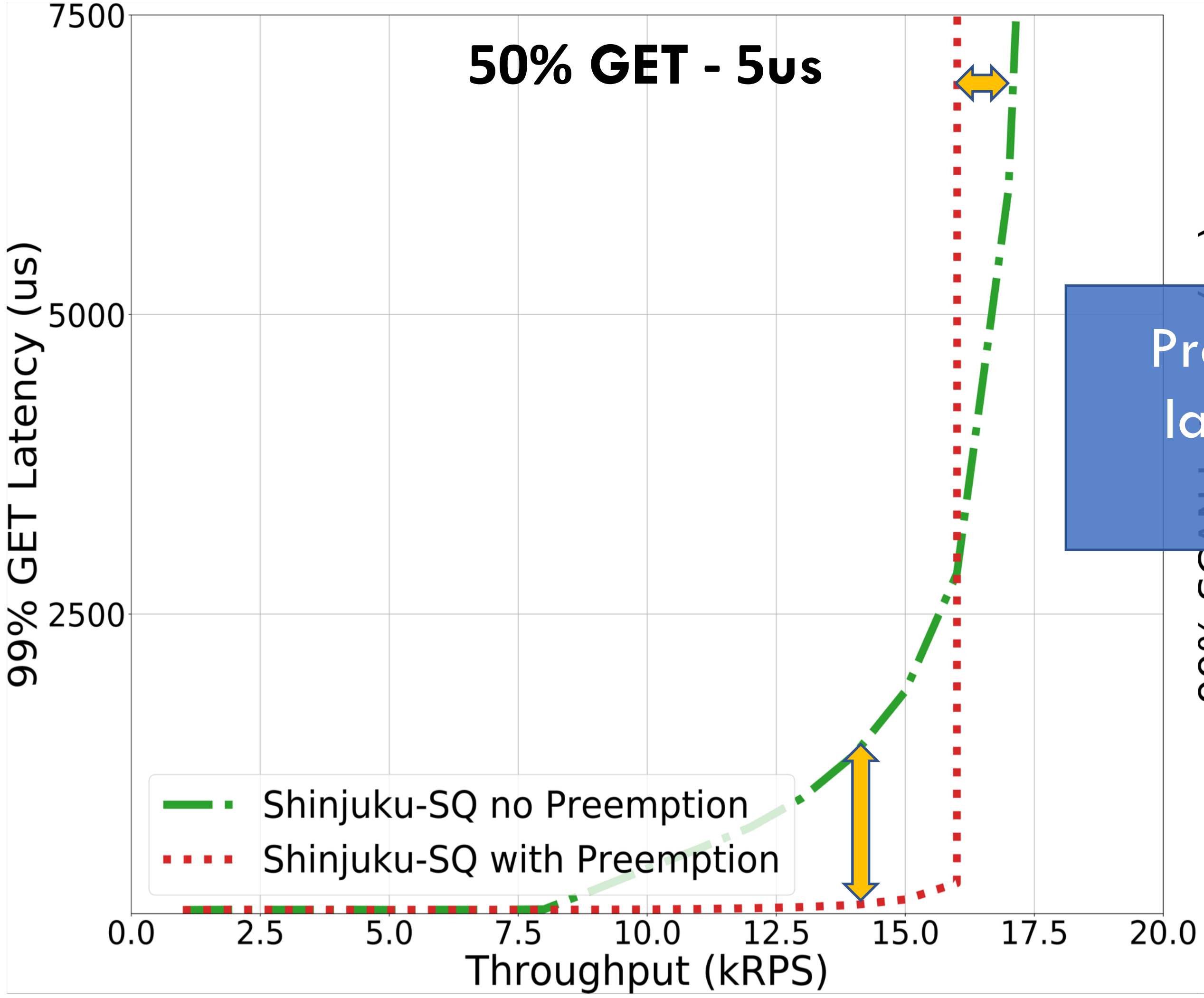
Single Queue no Preemption

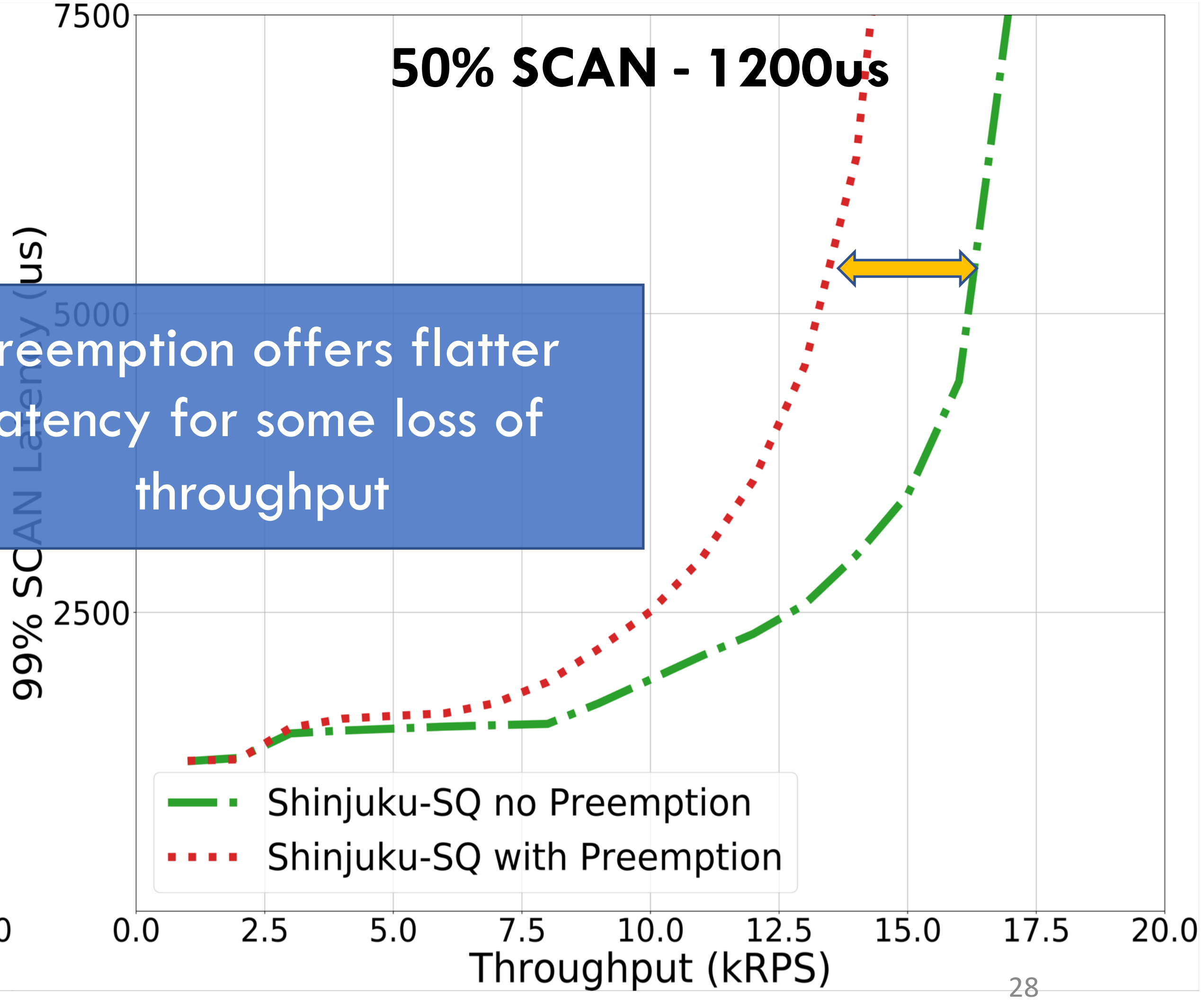




How important is each optimization?

Single Queue with Preemption

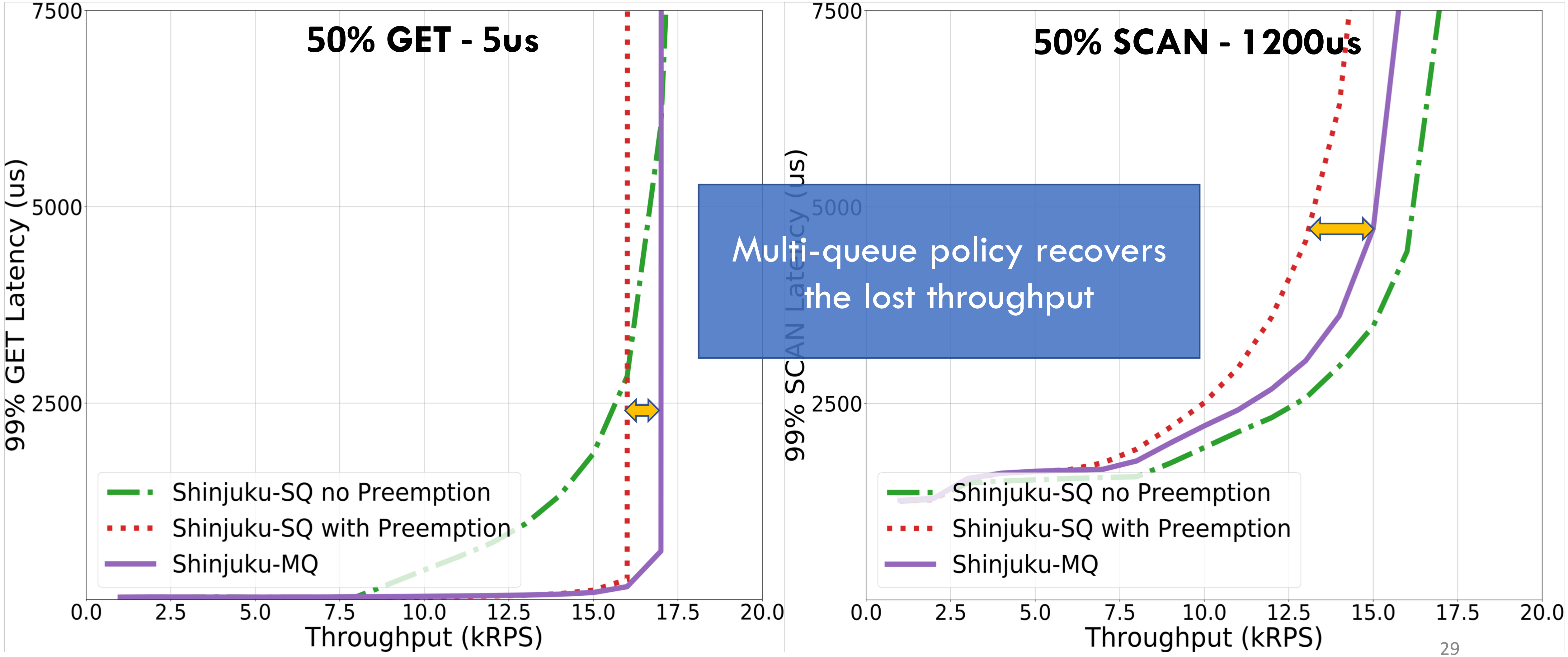




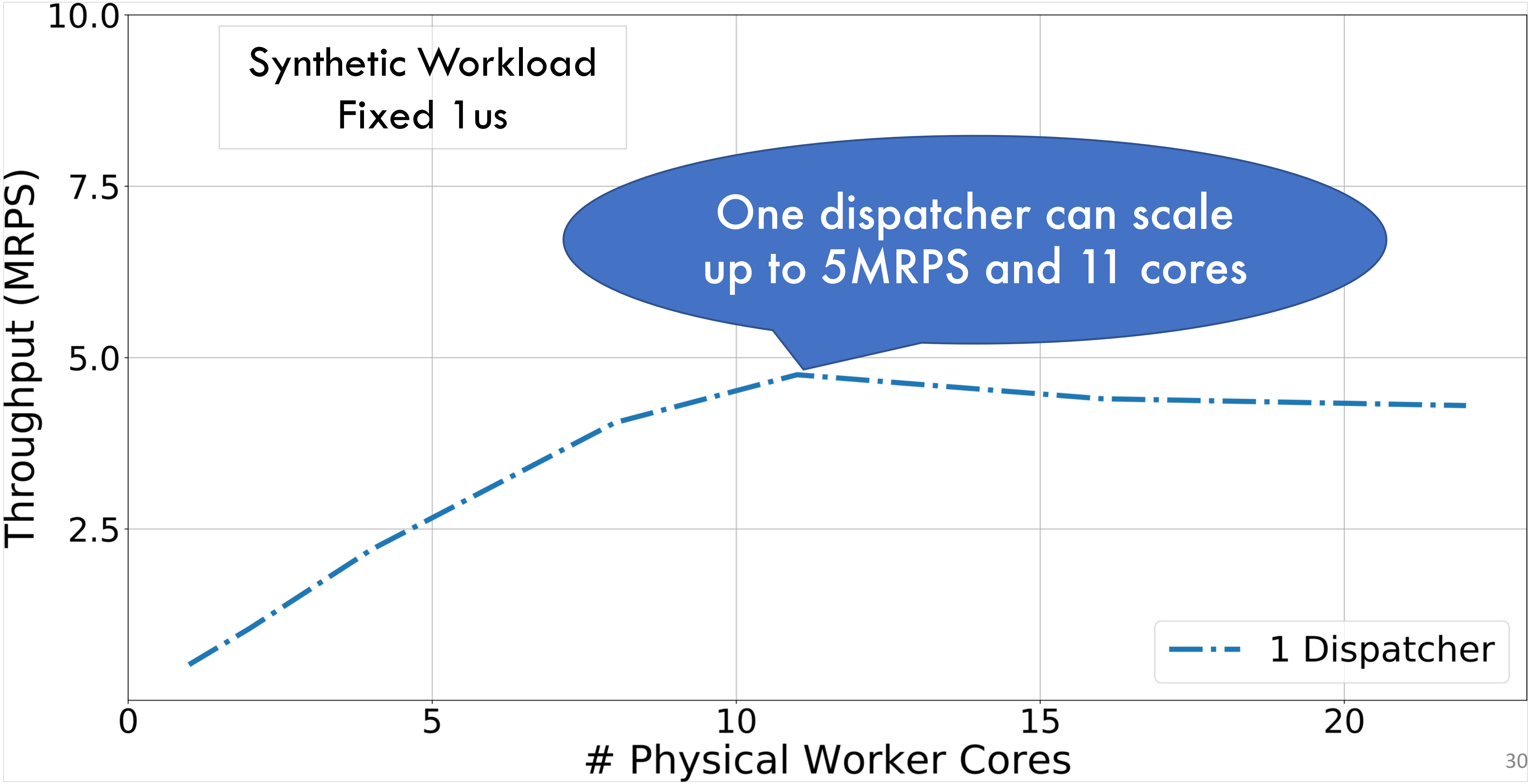
Preemption offers flatter latency for some loss of throughput

How important is each optimization?

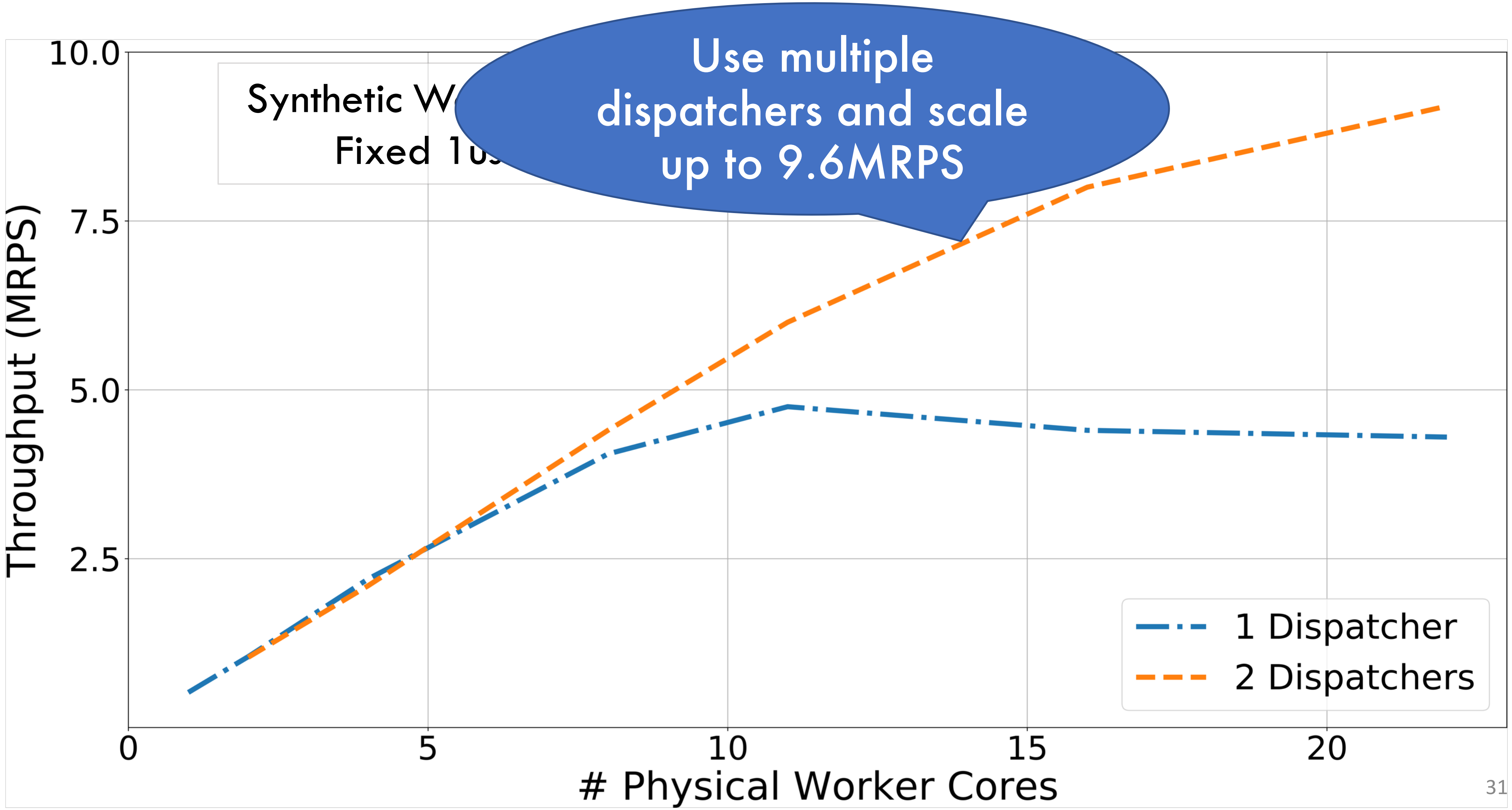
Multiple Queues with Preemption



Does Shinjuku scale?



Does Shinjuku scale?



More details in the paper

- Fast context switching
- How Shinjuku supports high line rates
- Placement policy of interrupted requests
- The problems of RSS-only scheduling of requests to cores
- More performance analysis

Conclusion

Low tail latency for general workloads requires:

- Preemptive Scheduling
- Centralized Queueing
- Flexible Scheduling Policies

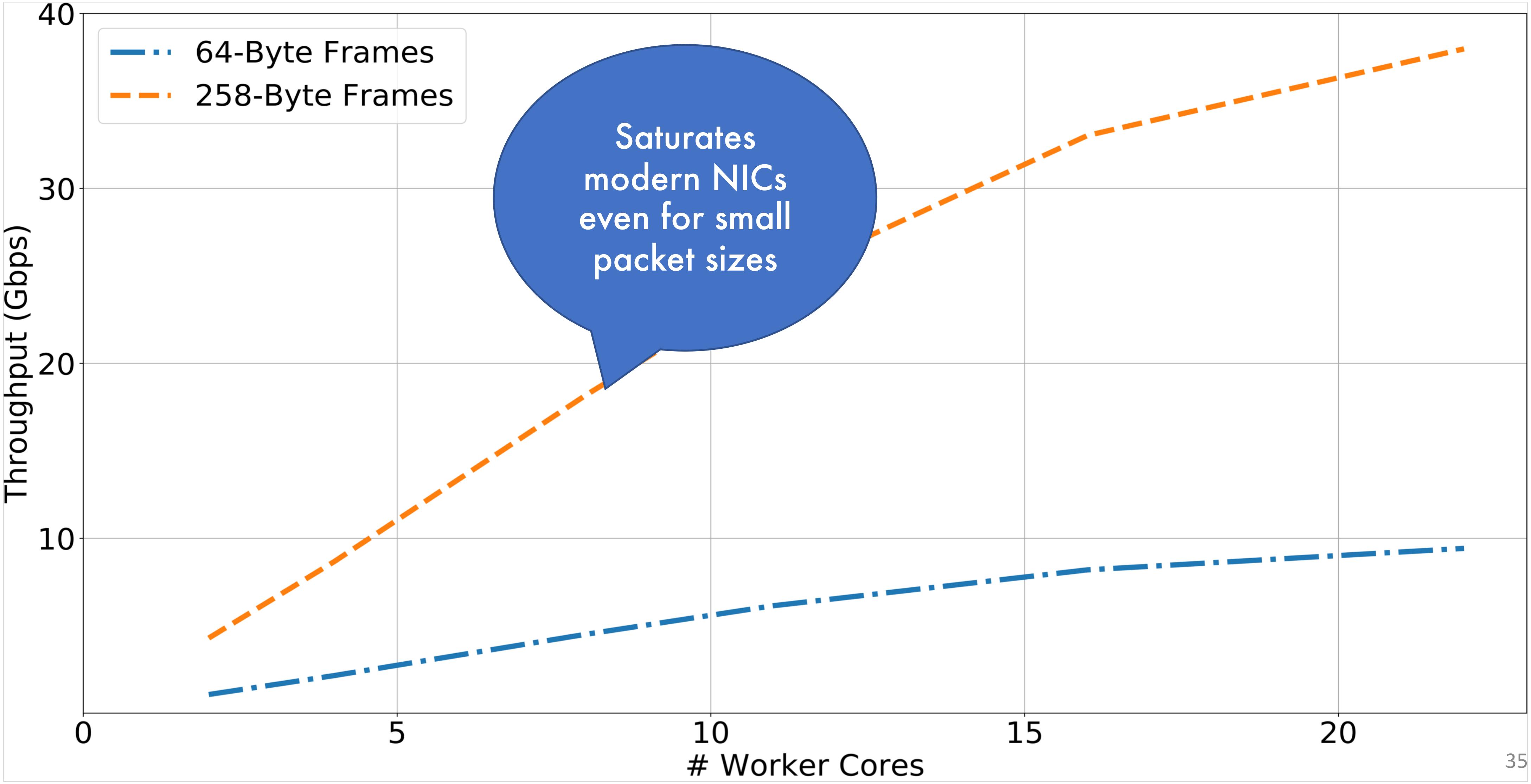
Shinjuku meets these demands at microsecond scale:

- Scalable centralized queue using dedicated core
- Preemption every 5us
- Latency-driven scheduling policies

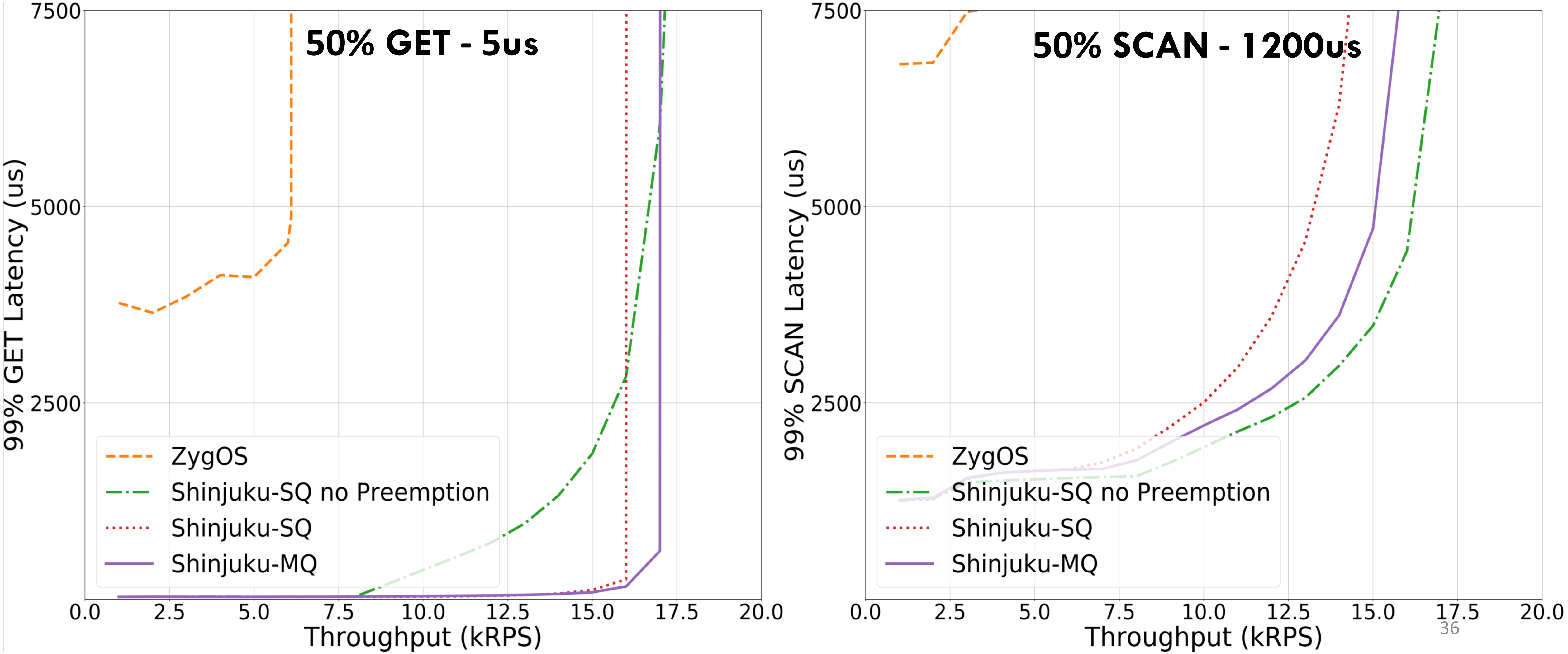
github.com/stanford-mast/shinjuku

Backup

Shinjuku Network Scaling



How important is each optimization?



Slowdown=
$$\frac{Total\ Latency}{Service\ Time}$$

Time slice matters