Size-aware sharding for improving tail latencies in in-memory key-value stores

Diego Didona (EPFL), Willy Zwaenepoel (University of Sydney)

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

ecocloud
an EPFL research center

THE UNIVERSITY OF
SYDNEY

Contributions

(1) **Size-aware sharding**

Improve tail latencies of in-memory key-value stores with heterogeneous item sizes

(2) **Minos in-memory key-value store**

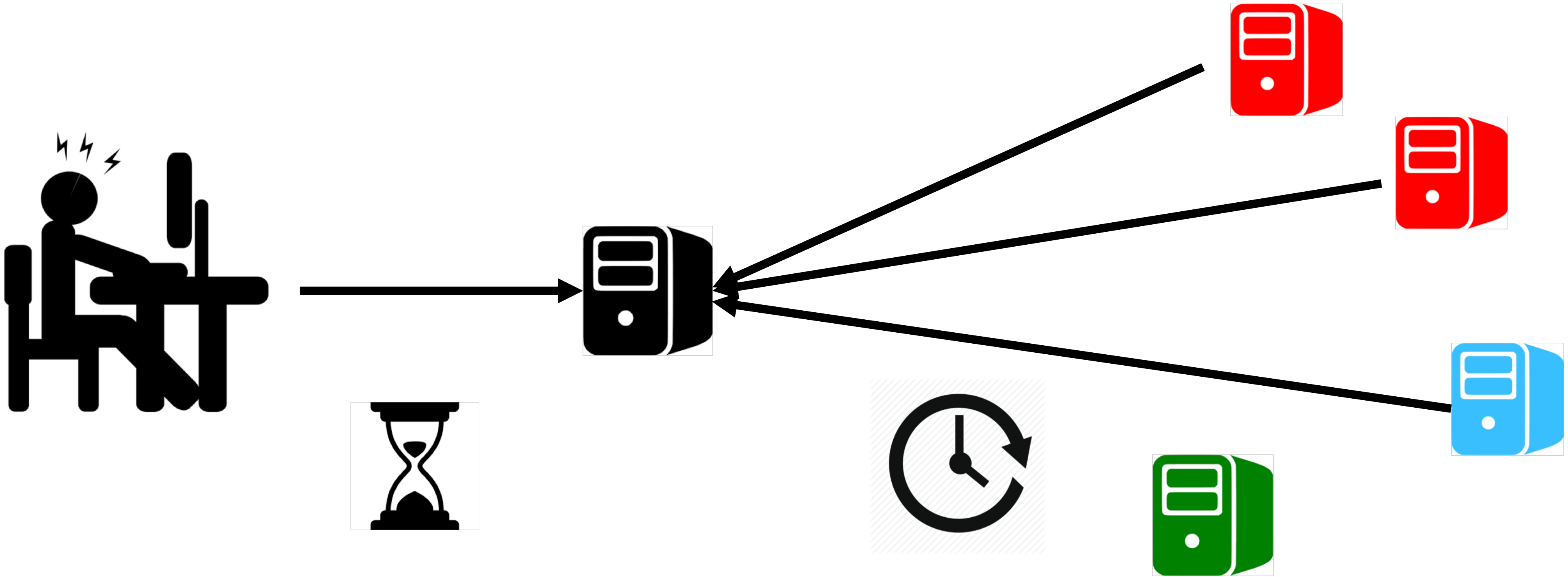
Order-of-magnitude lower 99th percentile latency

BACKGROUND

Tail latencies in high fan-out applications

Slowest reply determines request latency

SLO: N-th percentile of resp. time < X



In-memory key-value stores (KV)

- Widespread solution to deliver low latency

- Caches / non-persistent data repositories

State-of-the-art KVs: design

DPDK
DATA PLANE DEVELOPMENT KIT

High-bandwidth, multi-queue NICs
+
Kernel-bypassing network stacks

Run-to-completion model
+
Ad hoc data structures and CC

State-of-the-art KVs: performance

μsec-scale latencies @ several Mops/sec

State-of-the-art KVs: performance

μsec-scale latencies @ several Mops/sec

But high tail latencies with heterogeneous item sizes

Heterogeneous item sizes are common

Facebook [SIGMETRICS12]

Wikipedia [ISCA13]

Flickr [ISCA13]

Memcachier [NSDI19]

Heterogeneous item sizes are common

Facebook [SIGMETRICS12]

Wikipedia [ISCA13]

Flickr [ISCA13]

Memcachier [NSDI19]

Heavy tail: few large requests but very costly

Why high tail latencies?

1. Head-of-line blocking

2. Convoy effect

Head-of-line blocking

Small requests enqueued behind a large

Convoy effect

Burst of large requests may take most (or all) cores

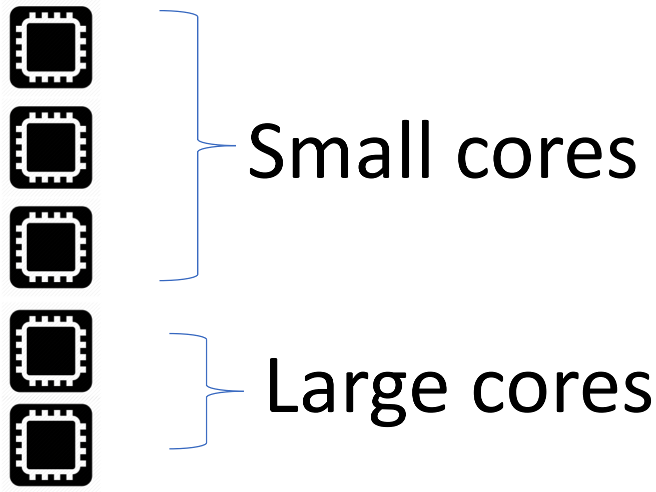
SIZE-AWARE SHARDING

Size-aware sharding

1. Small and large requests on disjoint sets of cores
   ➔Avoid head-of-line blocking

2. Reserve some cores for small requests
   ➔Avoid convoy effect

Size-aware sharding in operation



Small cores

Large cores

Size-aware sharding in operation

Small cores

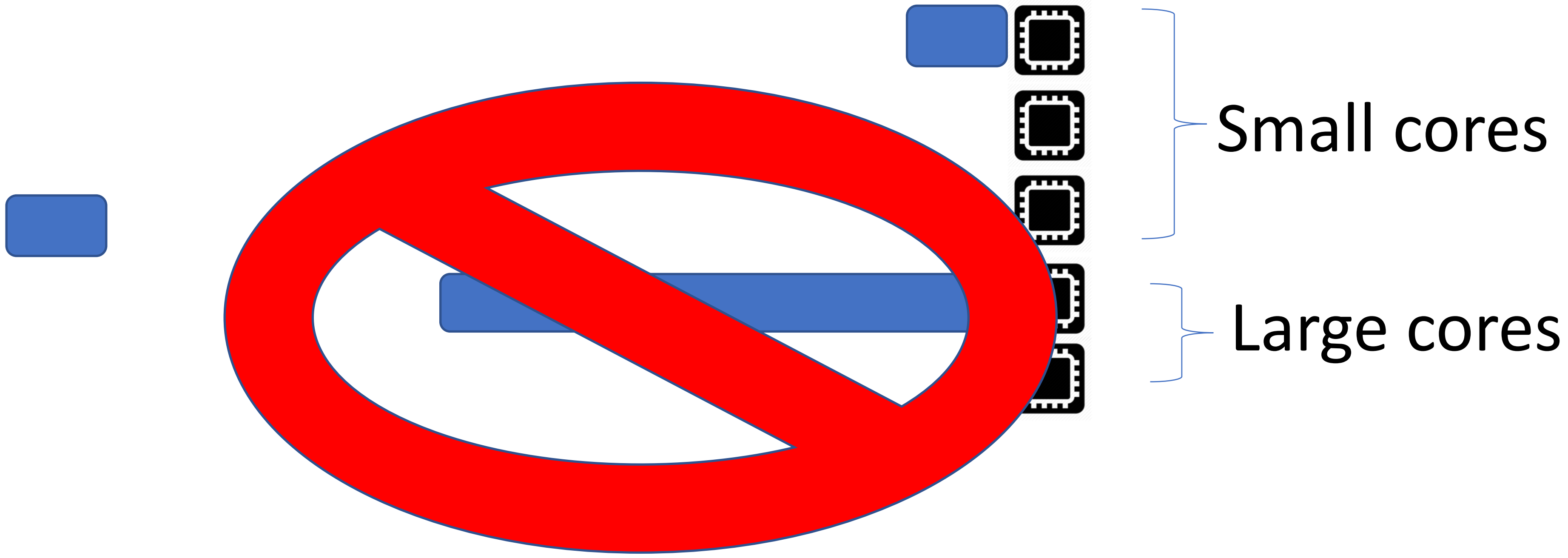
Large cores

Size-aware sharding in operation

Small cores

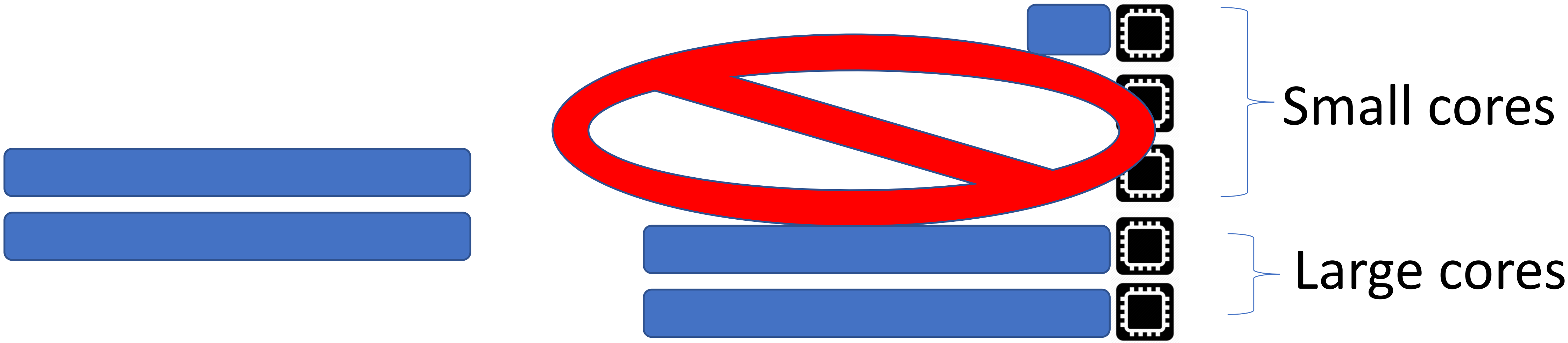
Large cores

Size-aware sharding in operation



Size-aware sharding in operation

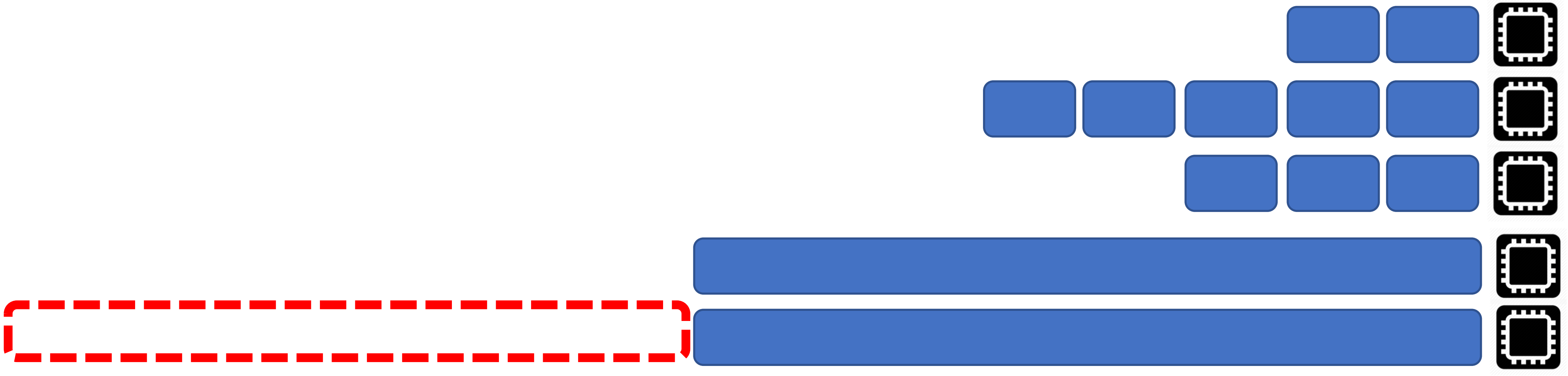
Size-aware sharding in operation



Size-aware sharding in operation

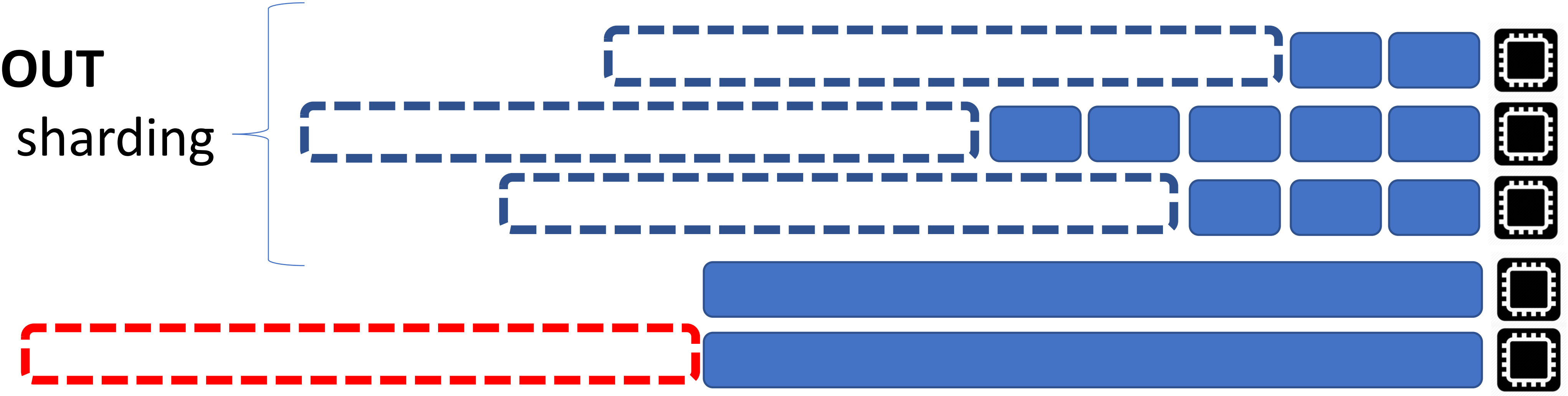
Trade-off: large requests take longer

Trade-off: large requests take longer



Trade-off: large requests take longer

WITHOUT
size-aware sharding



MINOS IN-MEMORY KV

Implementation

- Single-node, PUT-GET
- Commodity hardware
- No data durability

Minos design challenges

- Small vs large threshold
- Core partitioning
- Request dispatch

Minos design challenges

- Small vs large threshold
- Core partitioning
- Request dispatch

Main insight

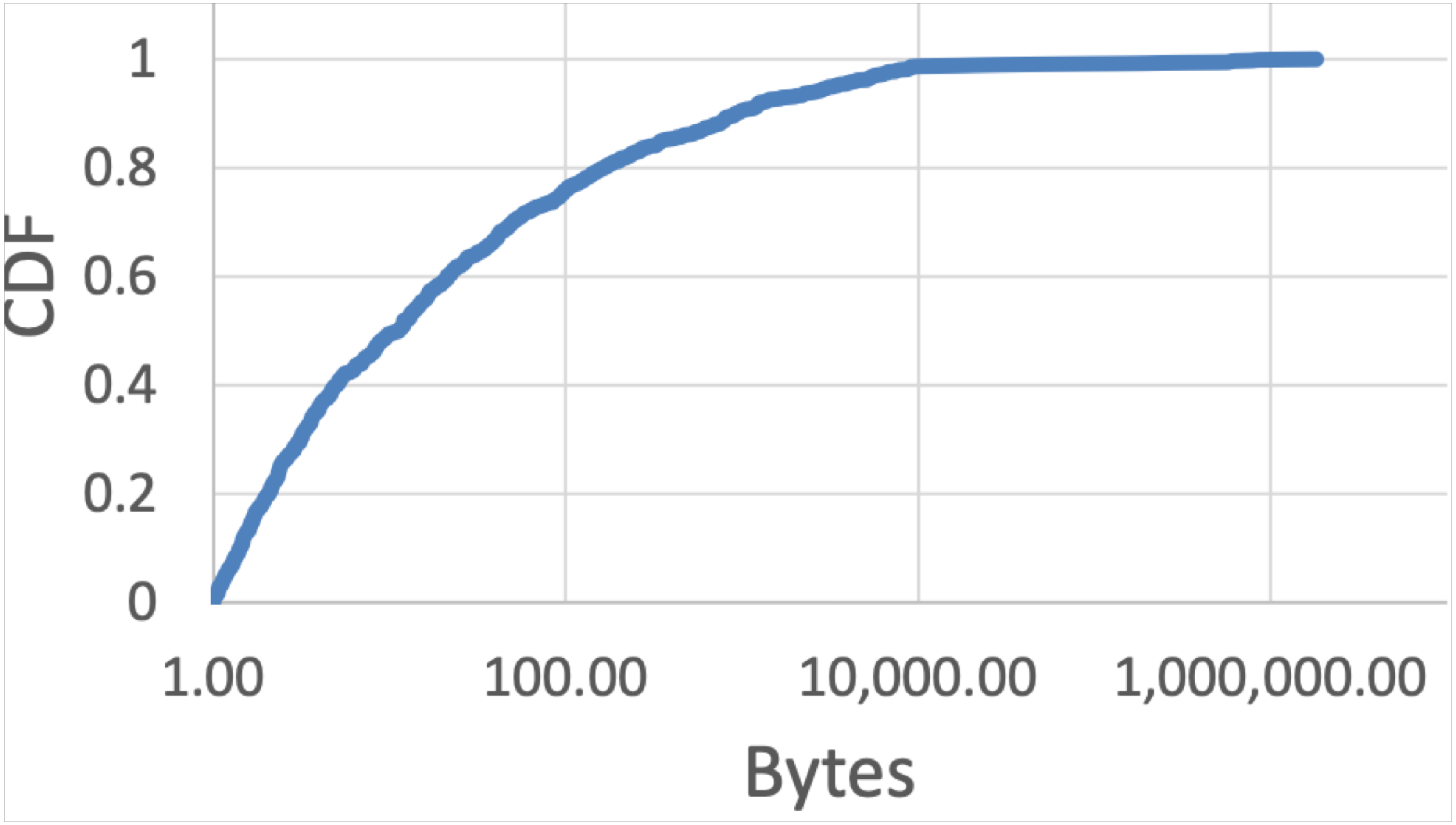
Improve **N**-th percentile of latencies

↓

Improve latencies of **N**% smallest requests

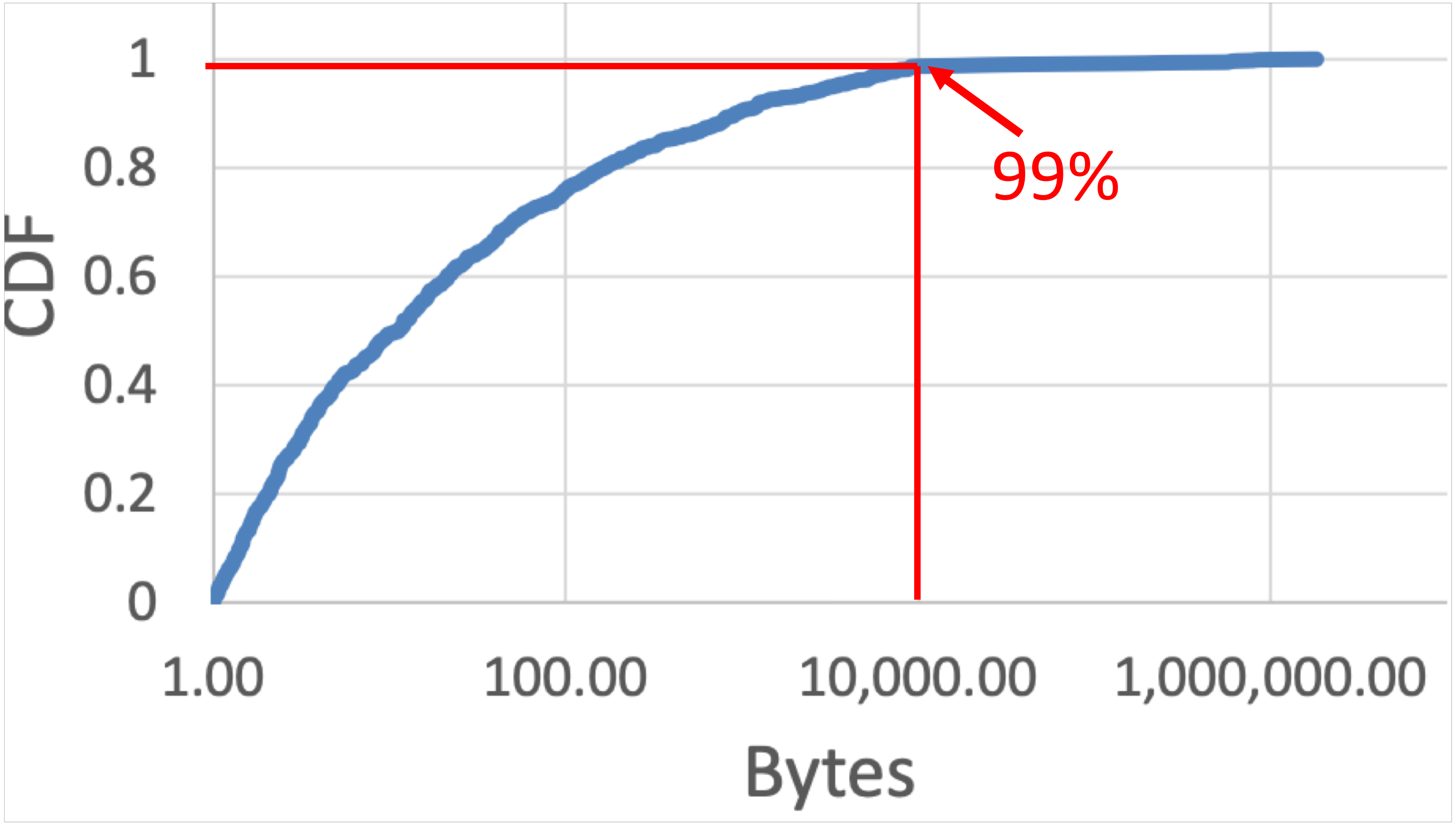
Example with 99th percentile

Obtain at runtime the CDF of the sizes of accessed items



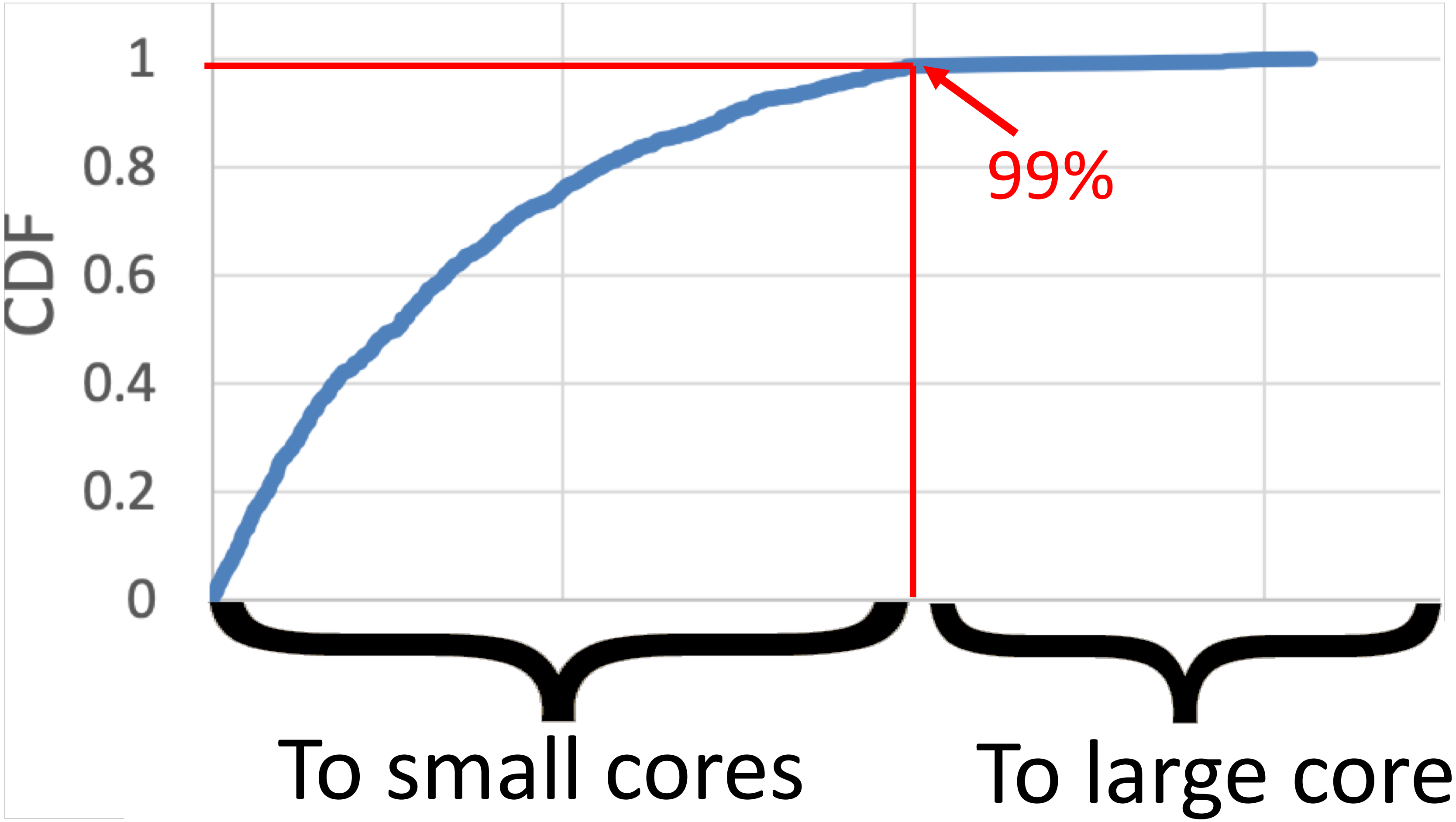
Example with 99th percentile

Obtain at runtime the CDF of the sizes of accessed items



Example with 99th percentile

Obtain at runtime the CDF of the sizes of accessed items



To small cores

To large cores

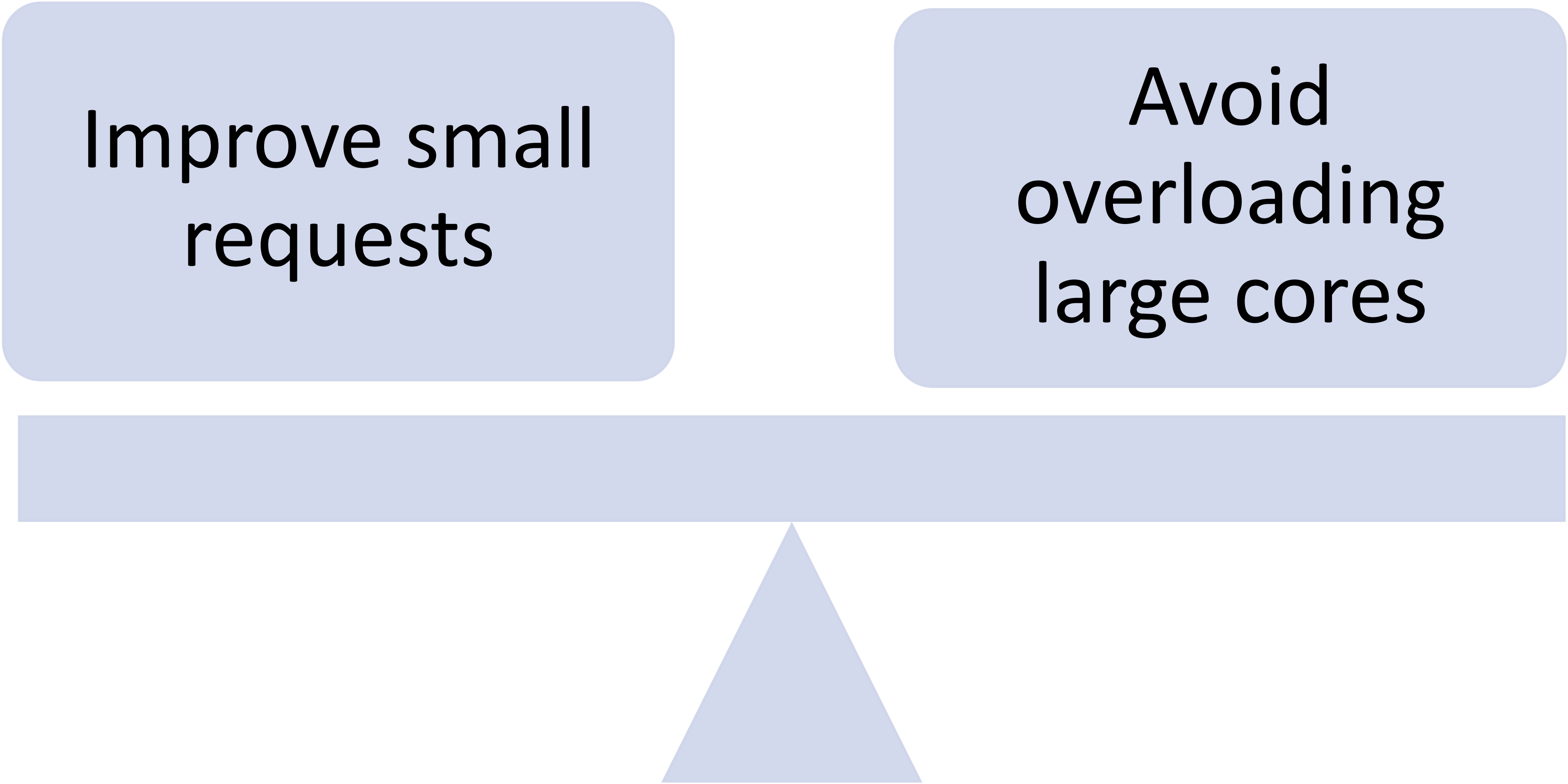
Minos design challenges

✓ Small vs large threshold

❓ Core partitioning

❓ Request dispatch

Goal



Improve small requests

Avoid overloading large cores

Load-proportional core allocation

K% of the load for small requests

↓

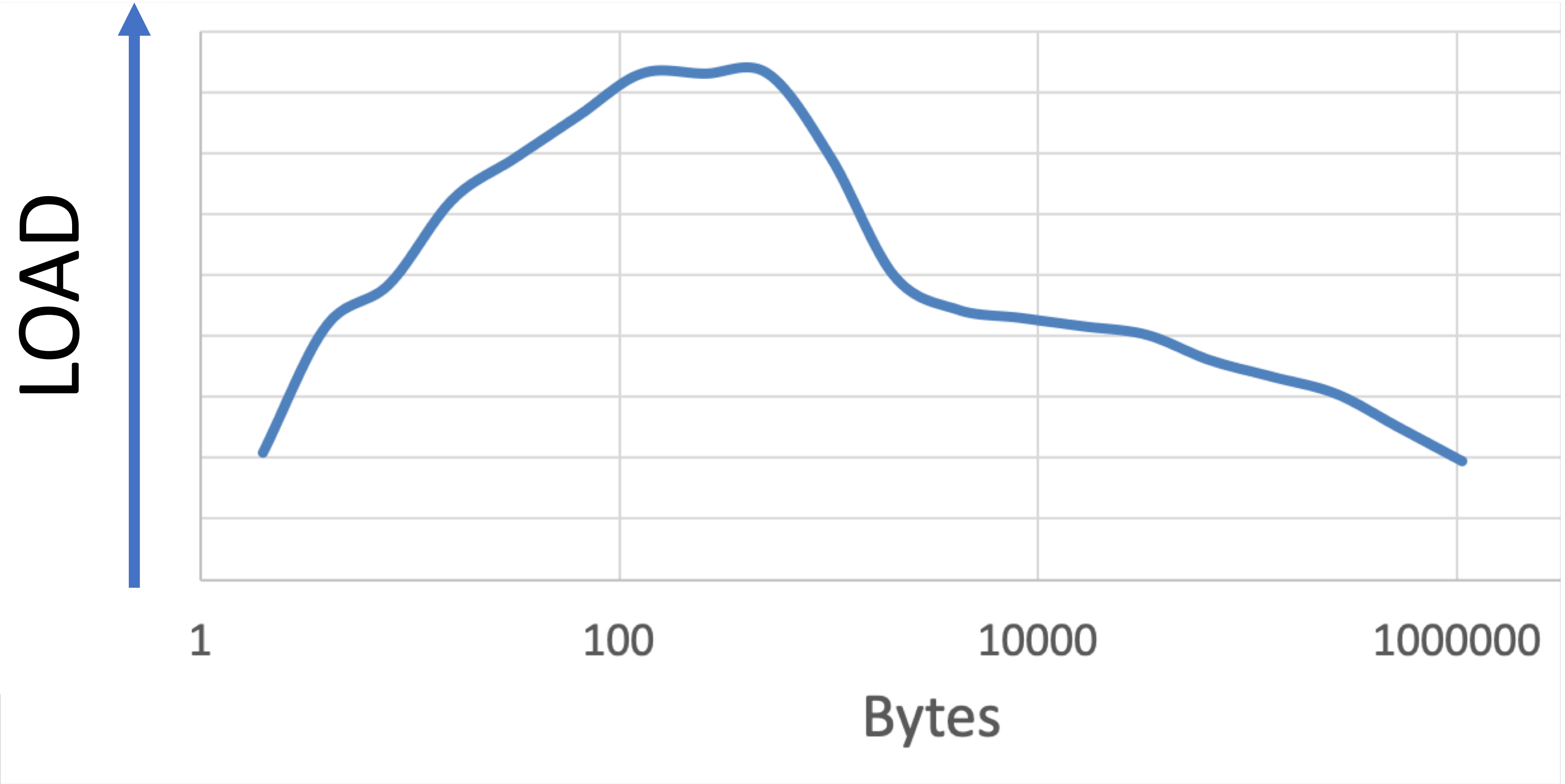
K% of small cores

Measuring the load of a request

Load of a request = # processed network packets

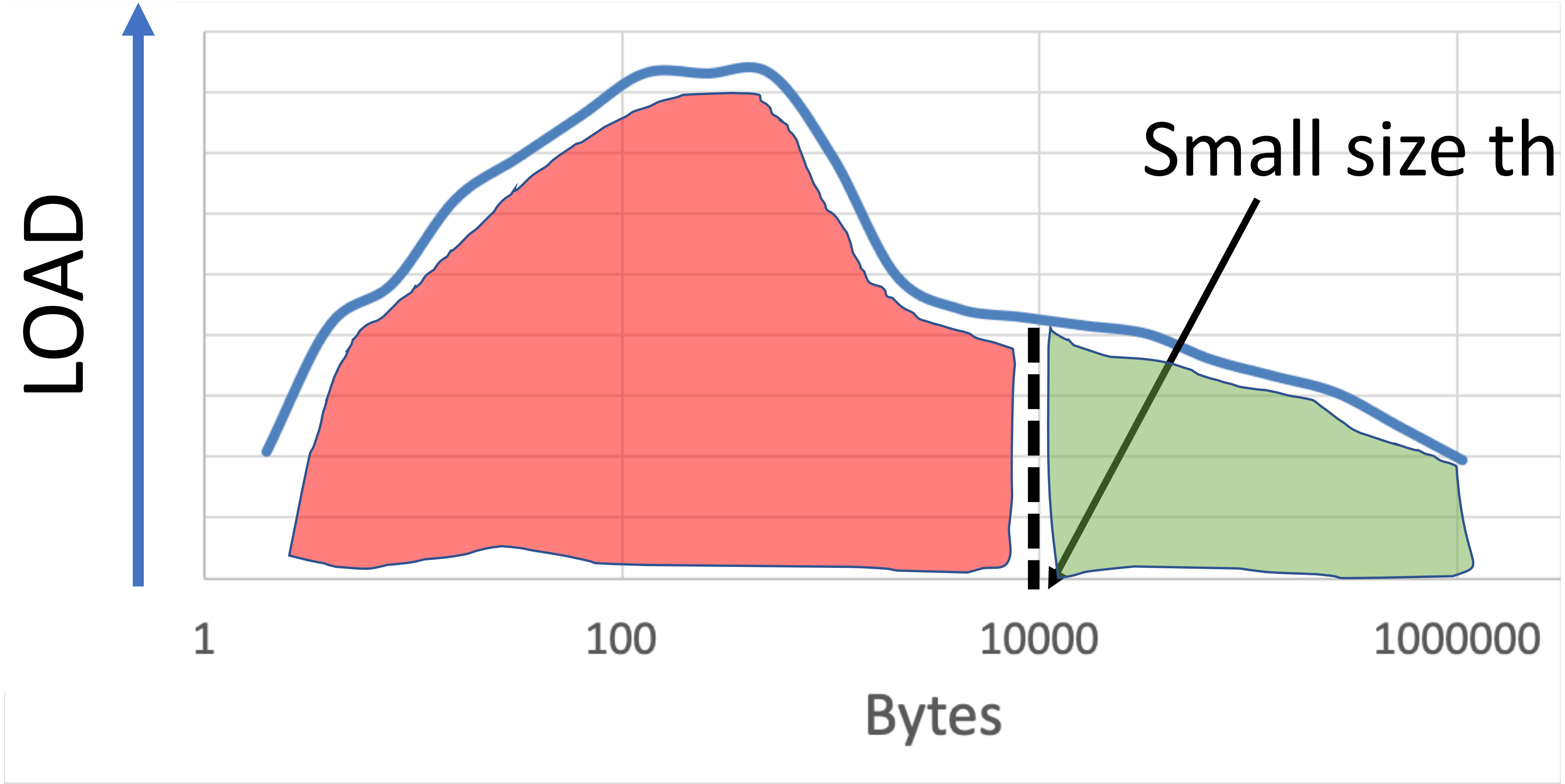
Dynamic core allocation

1. Obtain at runtime the load of requests of different sizes



Dynamic core allocation

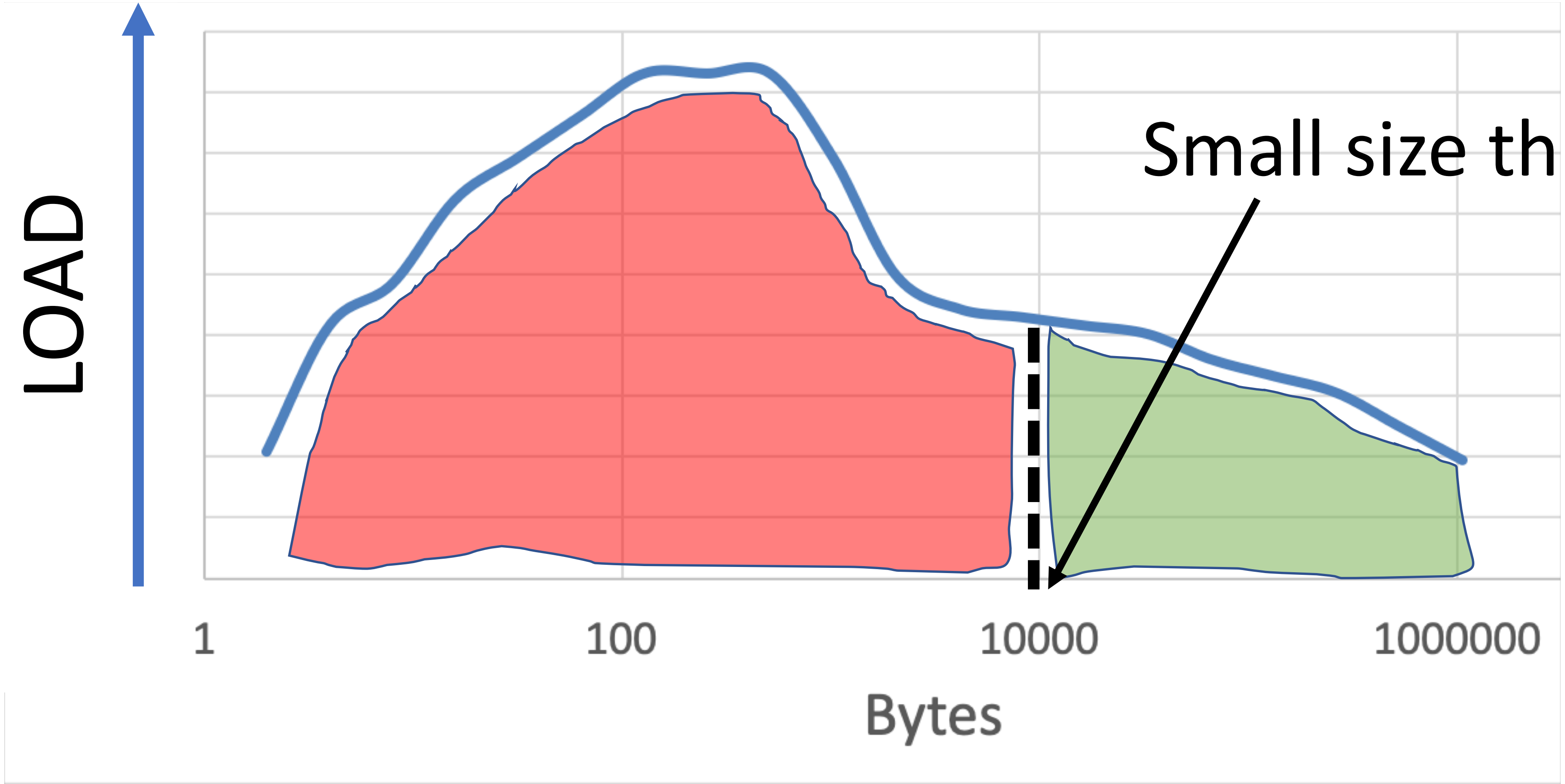
2. Fraction of small request load = 🟥 / (🟩 + 🟥)



Small size threshold

Dynamic core allocation

3. # Small cores = ceiling (small load * # total cores)

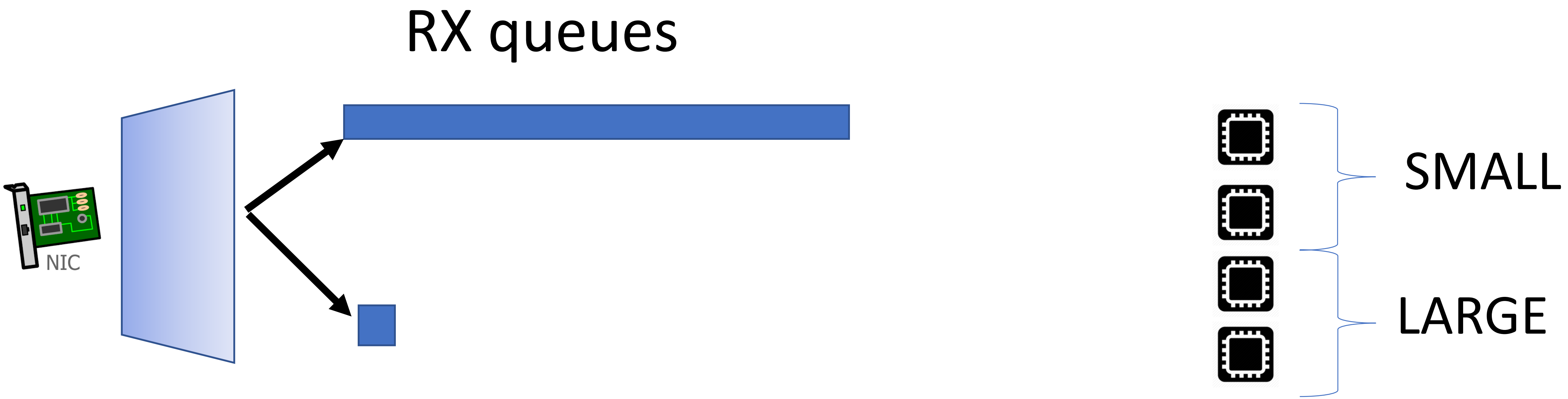


Small size threshold

Minos design challenges

- ✓ Small vs large threshold
- ✓ Core partitioning
- ? Request dispatch

Request size unknown *a priori*



Reduce software dispatch

Only small cores read from the NIC

1. From its own RX queue

2. From large cores’ RX queues

Operation of small cores on a request

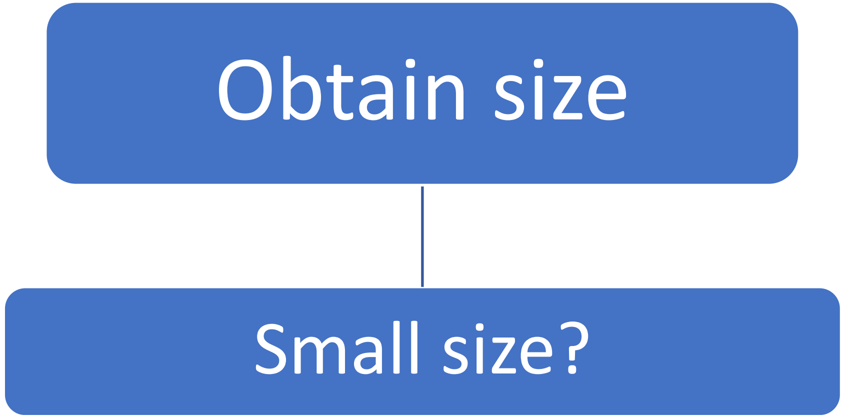
Obtain size

Operation of small cores on a request

Obtain size

GET: do lookup
PUT: in header

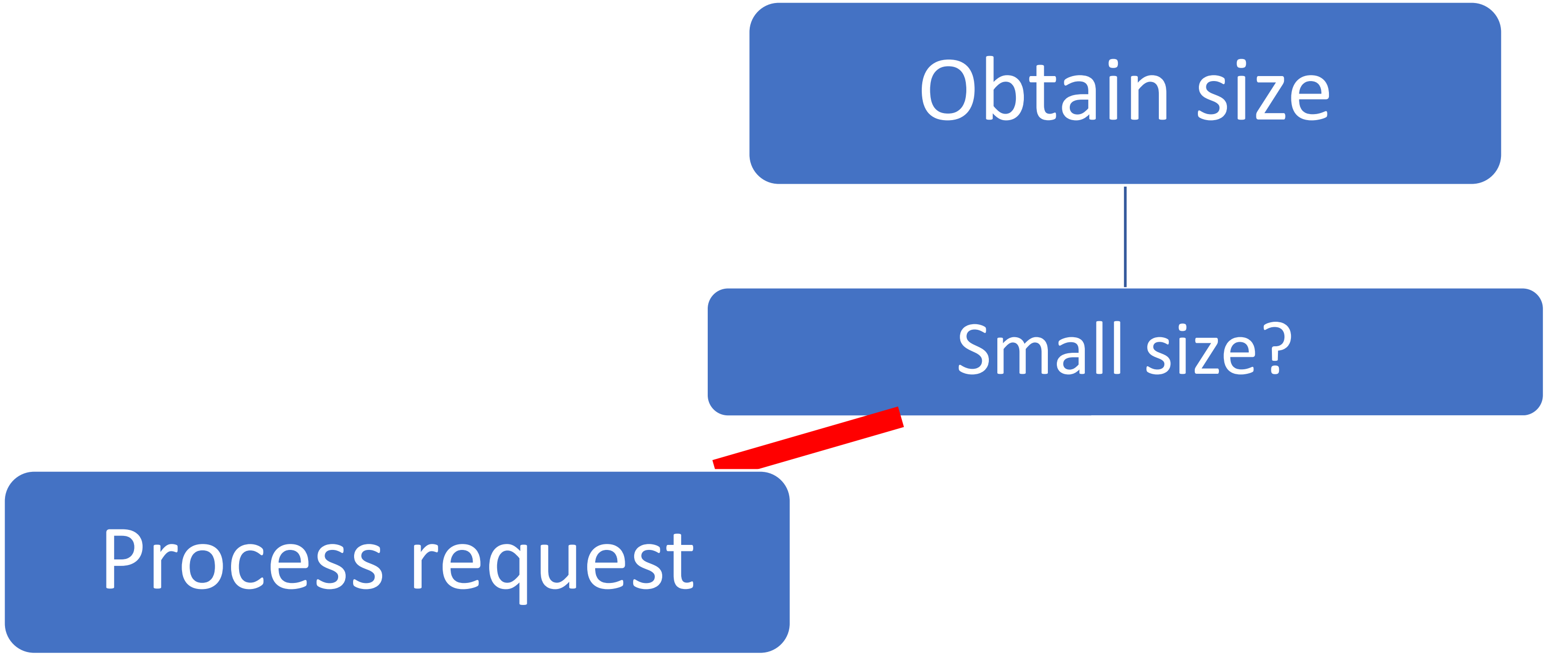
Operation of small cores on a request



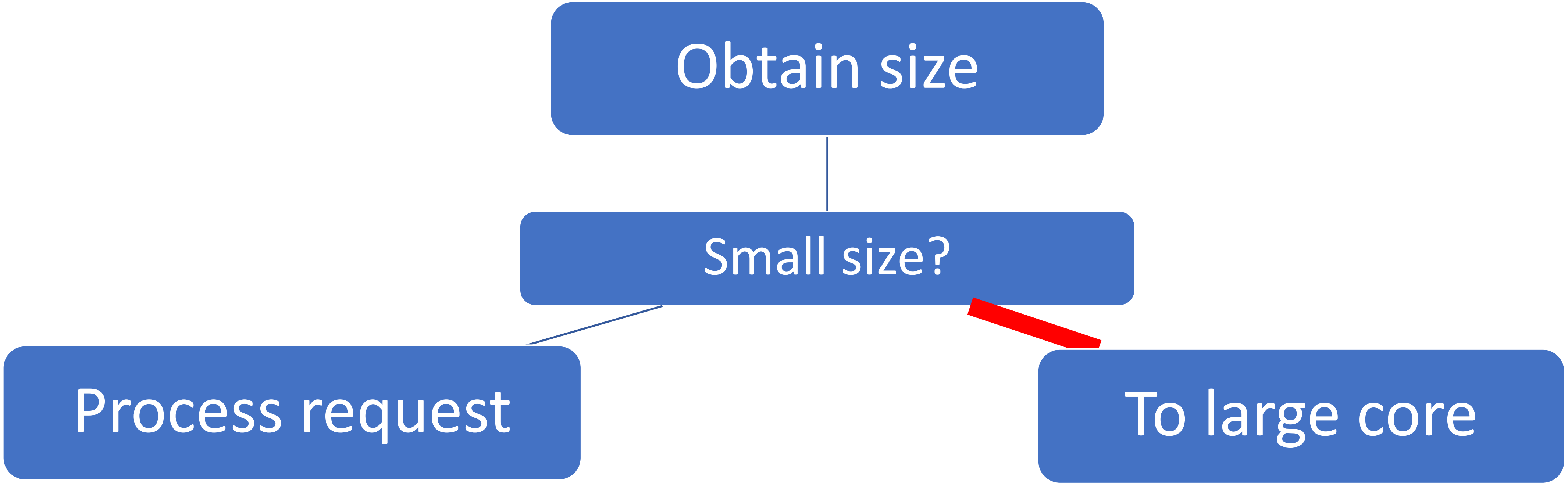
Obtain size

Small size?

Operation of small cores on a request



Operation of small cores on a request



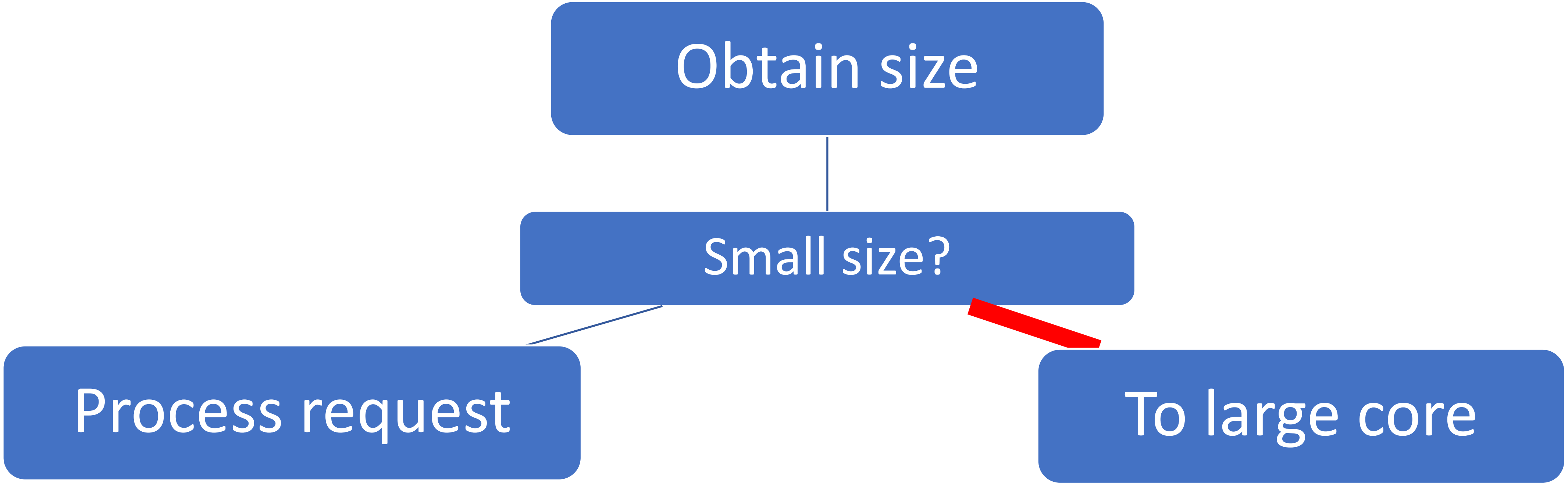
Obtain size

Small size?

Process request

To large core

Operation of small cores on a request



Obtain size

Small size?

Process request

To large core

SOFTWARE DISPATCH ONLY FOR FEW LARGE

EVALUATION

Test-bed

- Server: 8 cores, 40Gbps NIC, DPDK stack
- Wkld ~ ETC Facebook [SIGMETRICS12]
    - < 1 % large requests [1.5, 500] KB
- 95:5 GET:PUT ratio
- Skewed accesses (zipf 0.99)

Competitors

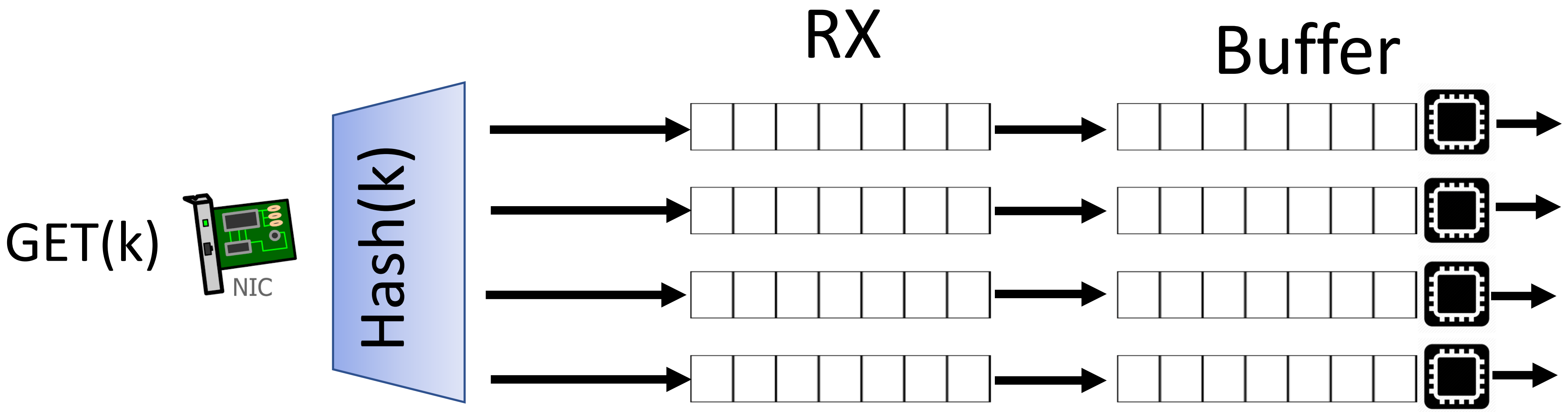
1. Early binding (~ MICA [NSDI14])

2. Early binding + stealing (~ ZygOS [SOSP17])

3. Late binding (~RAMCloud [TOCS15])

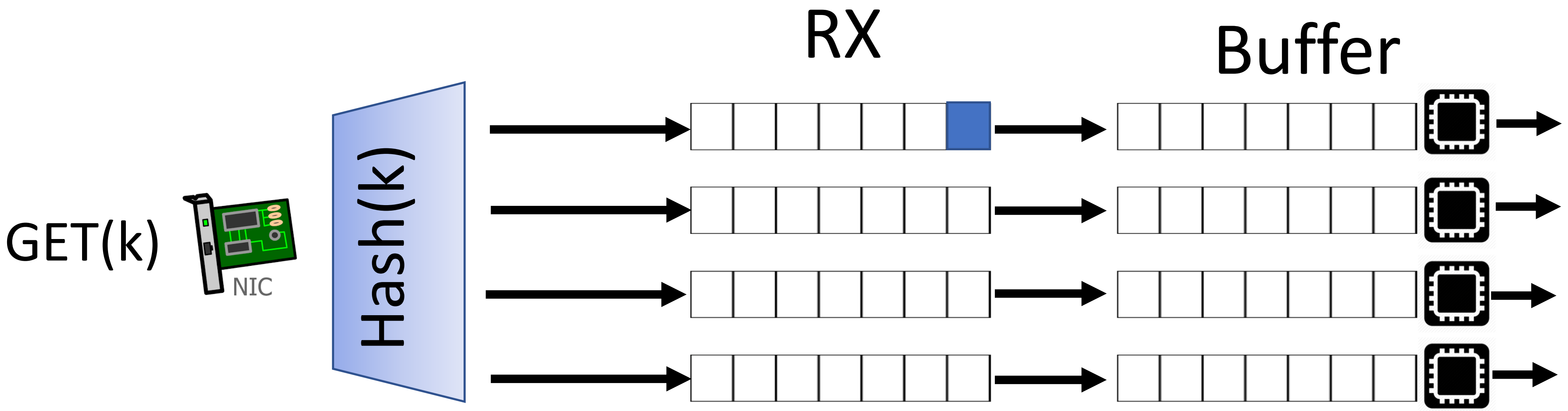
Early binding (MICA, NSDI2014)

Request -> core based on key-hash of target item



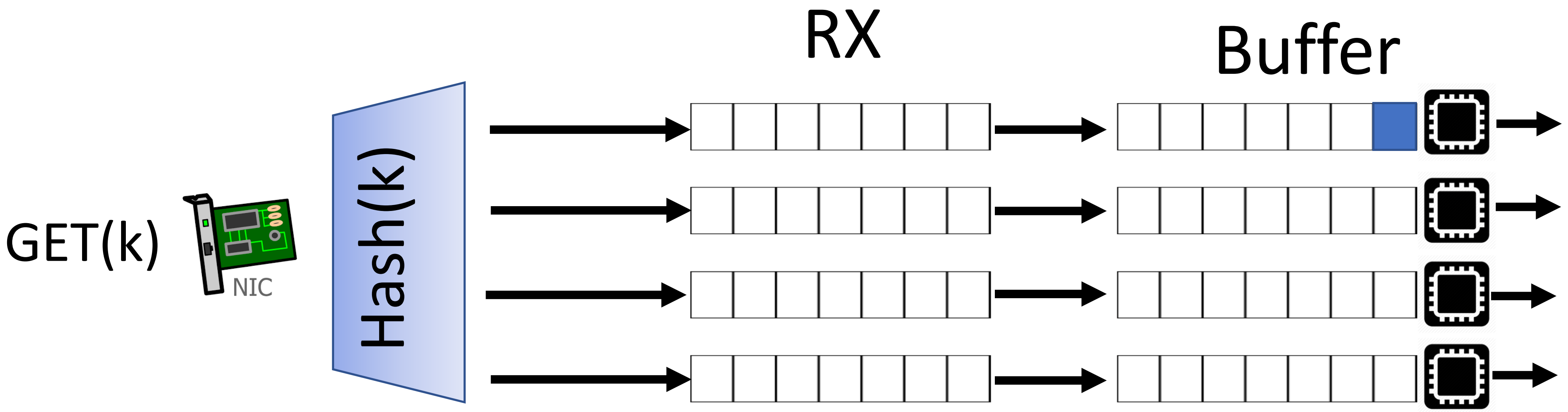
Early binding (MICA, NSDI2014)

Request -> core based on key-hash of target item



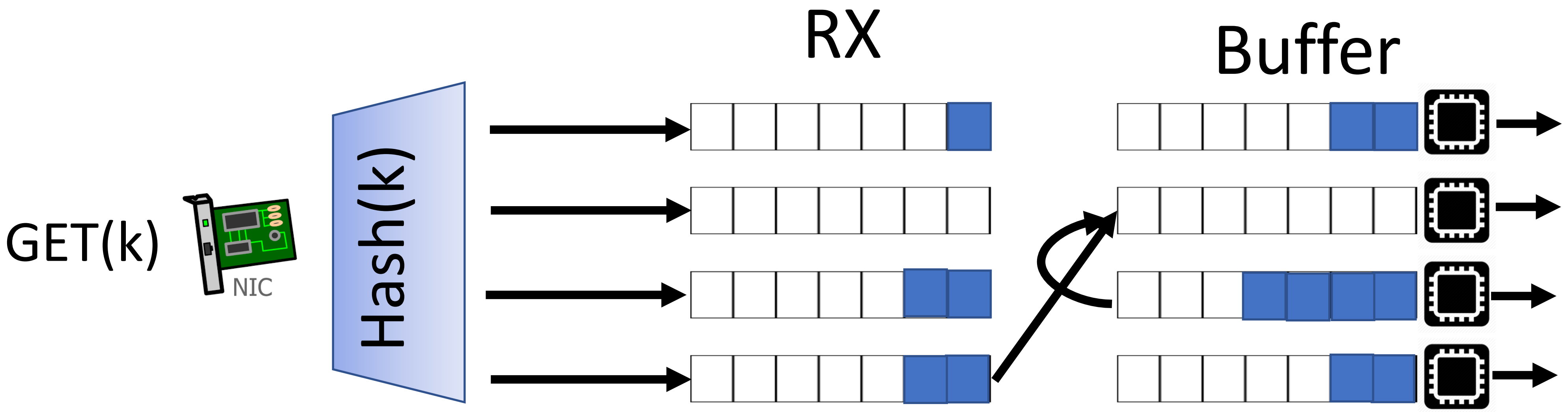
Early binding (MICA, NSDI2014)

Request -> core based on key-hash of target item



Early binding + stealing (ZygOS SOSP17)

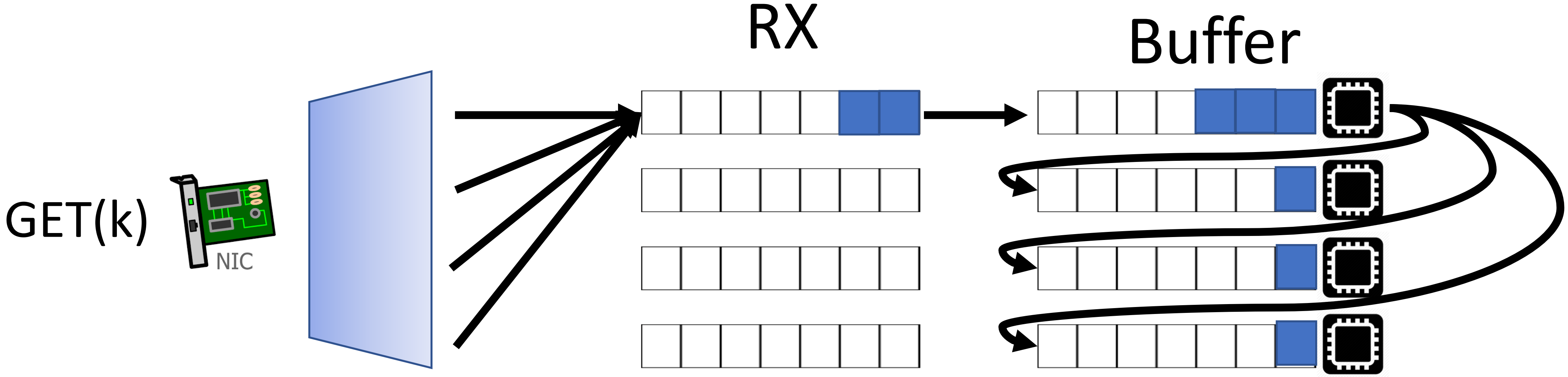
Idle cores steal requests from other queues/buffers



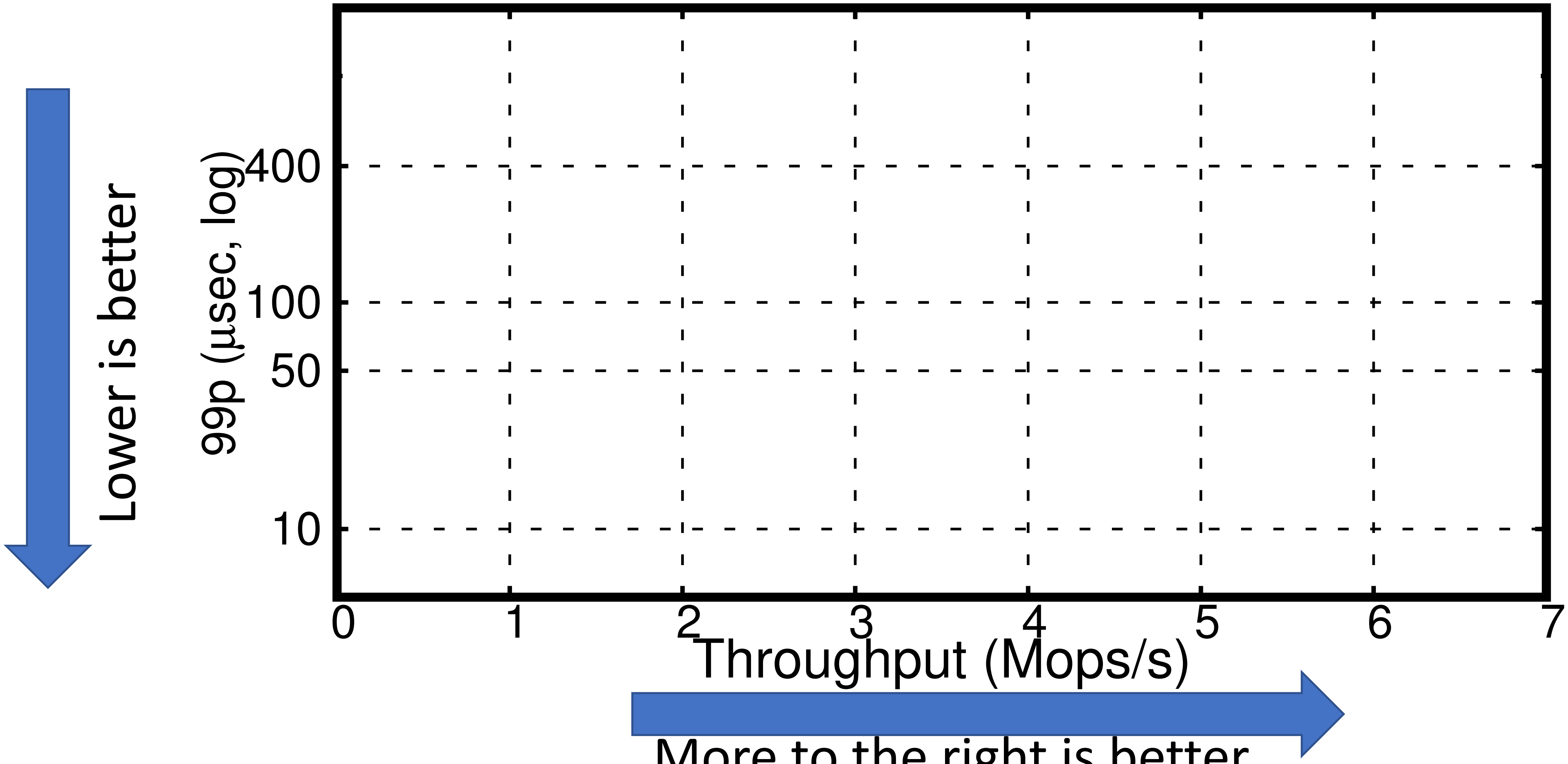
Late binding (RAMCloud TOCS15)

One core receives all requests

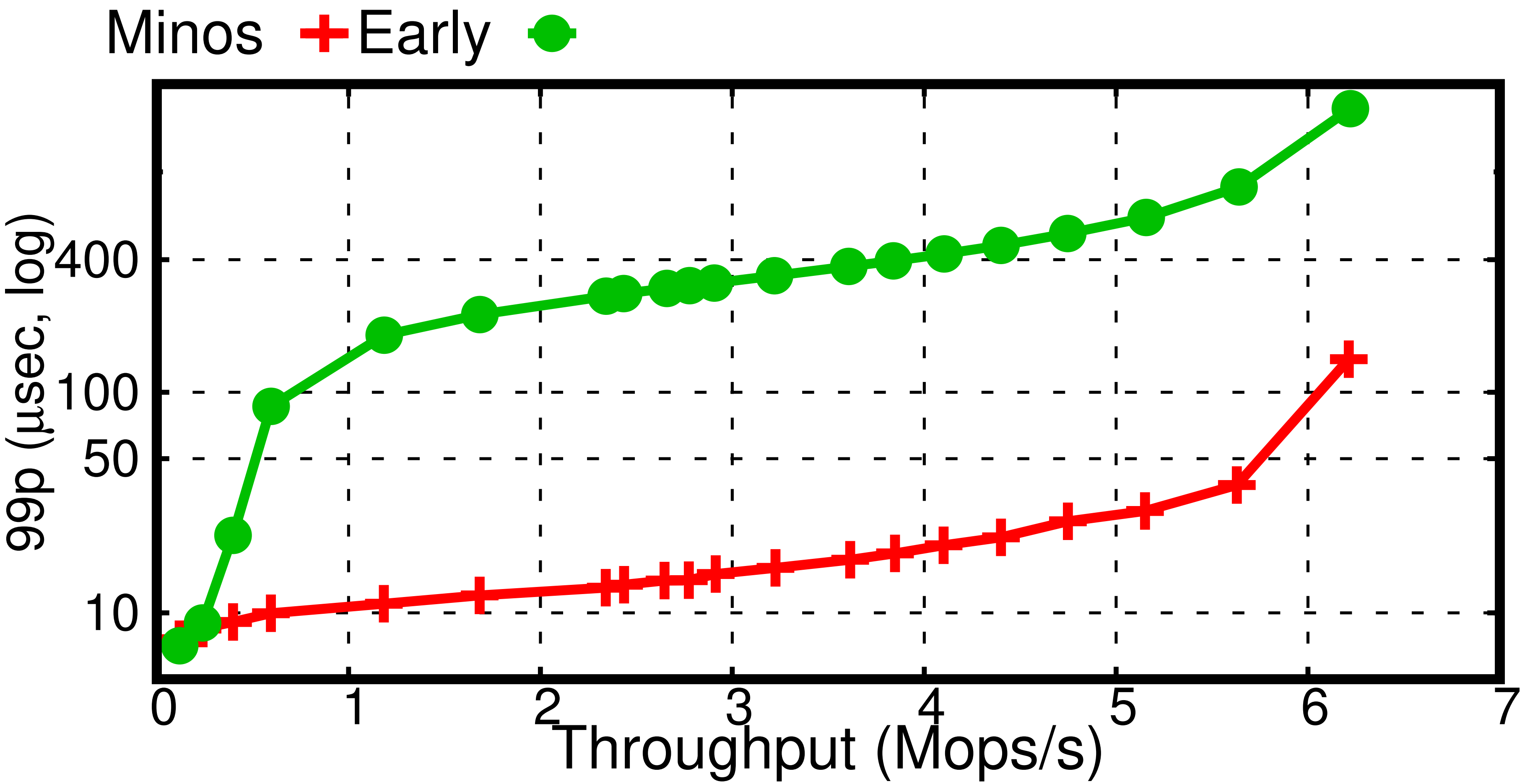
dispatches them to idle cores



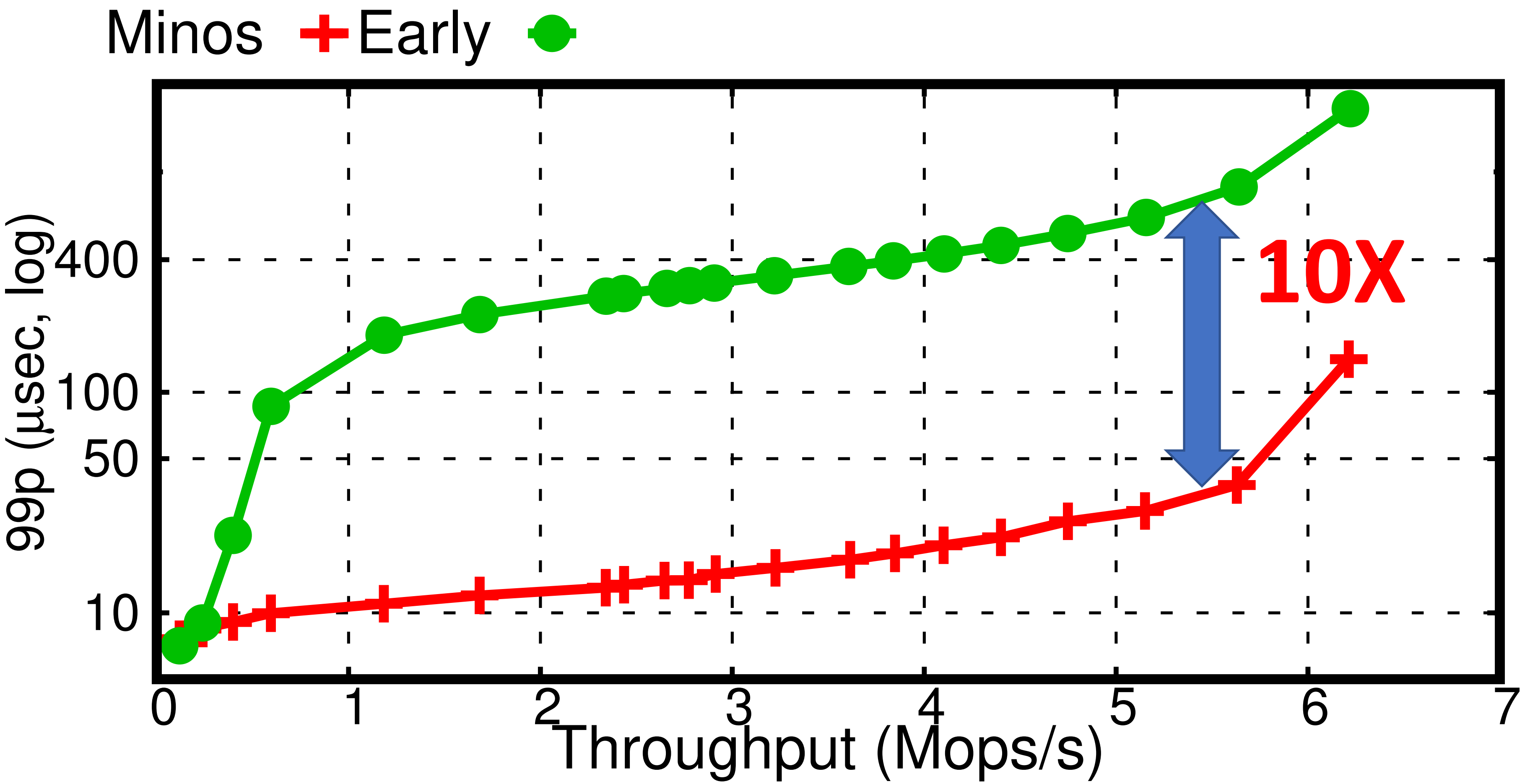
Throughput vs overall 99th latency



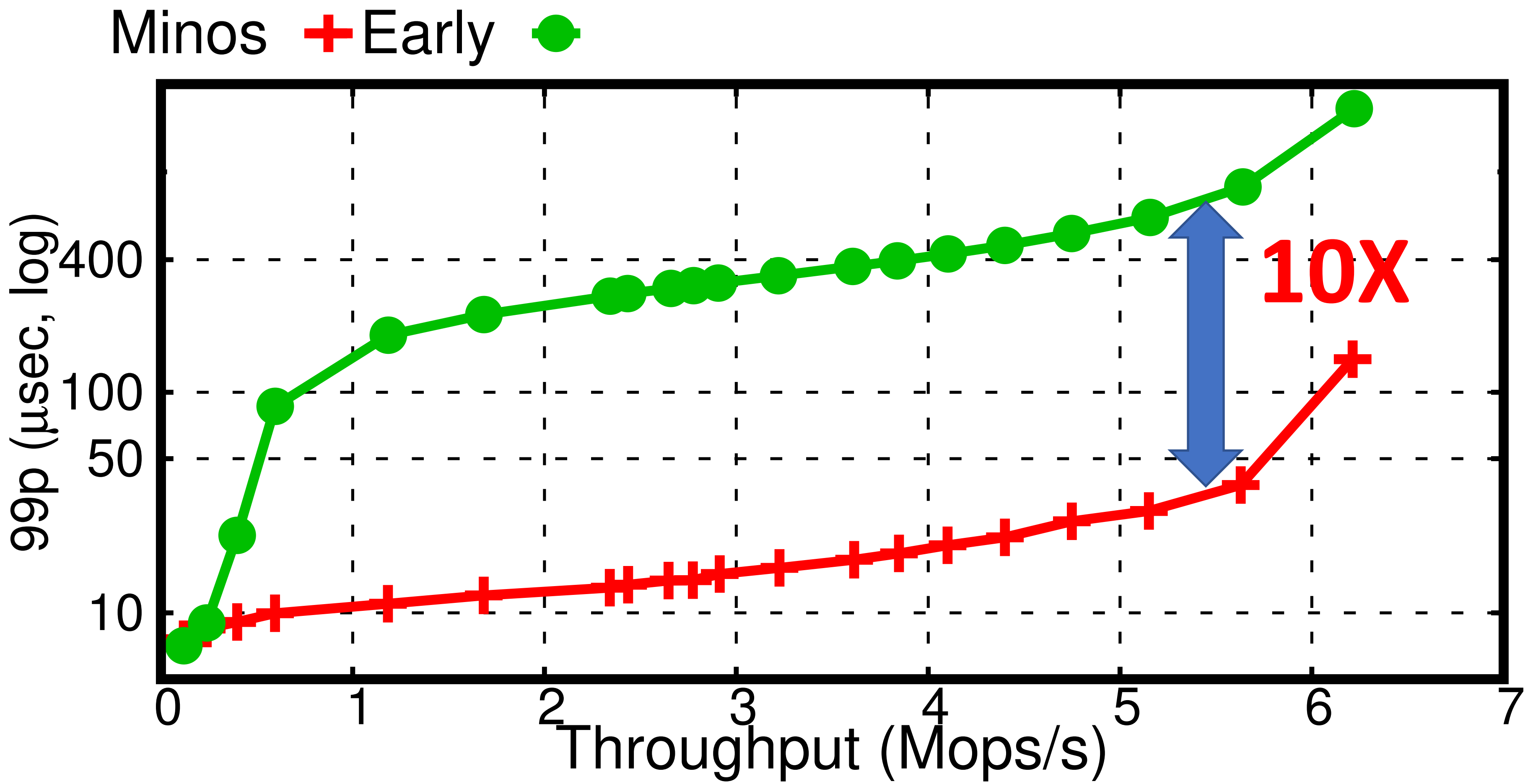
Minos vs early binding



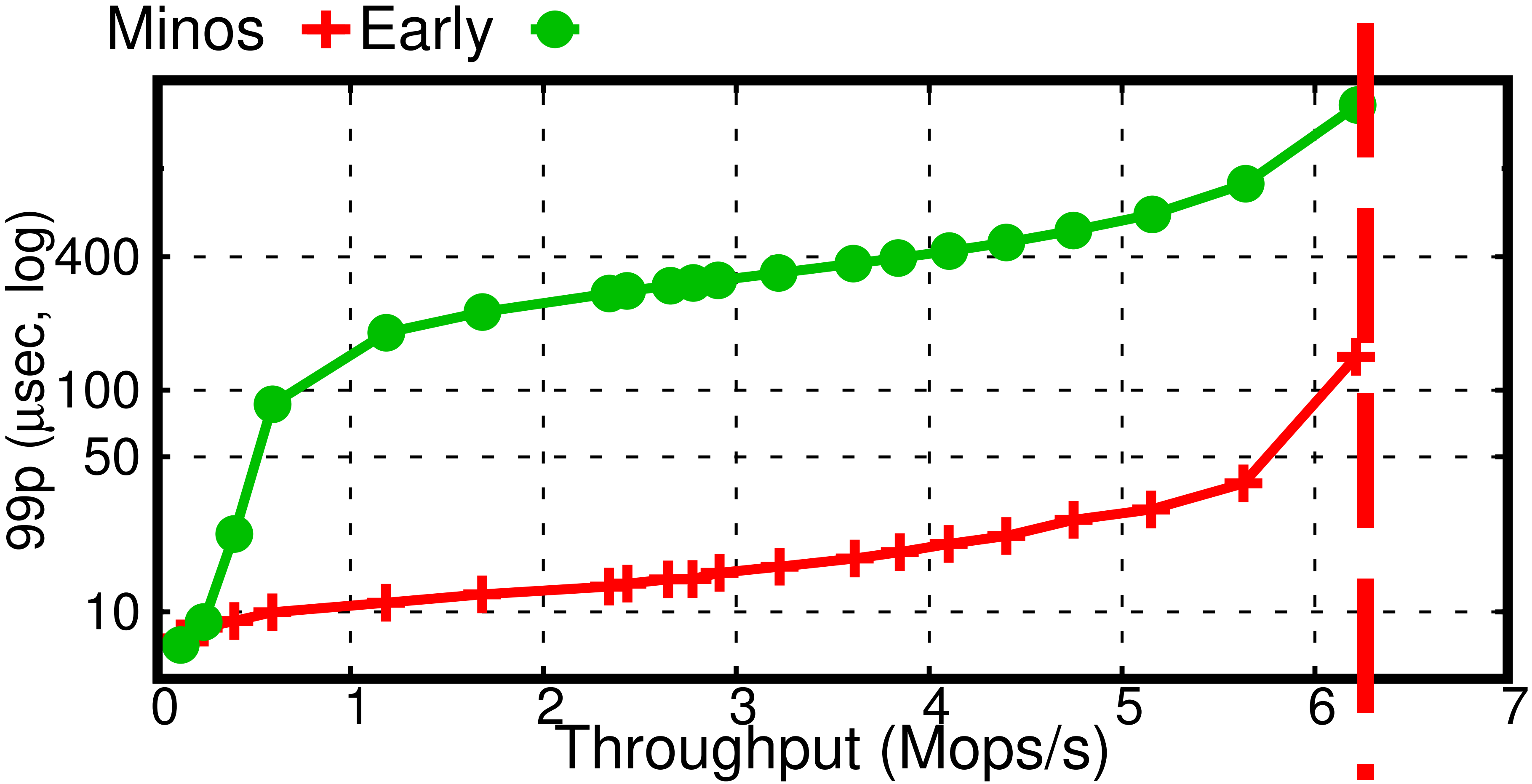
Lower latency



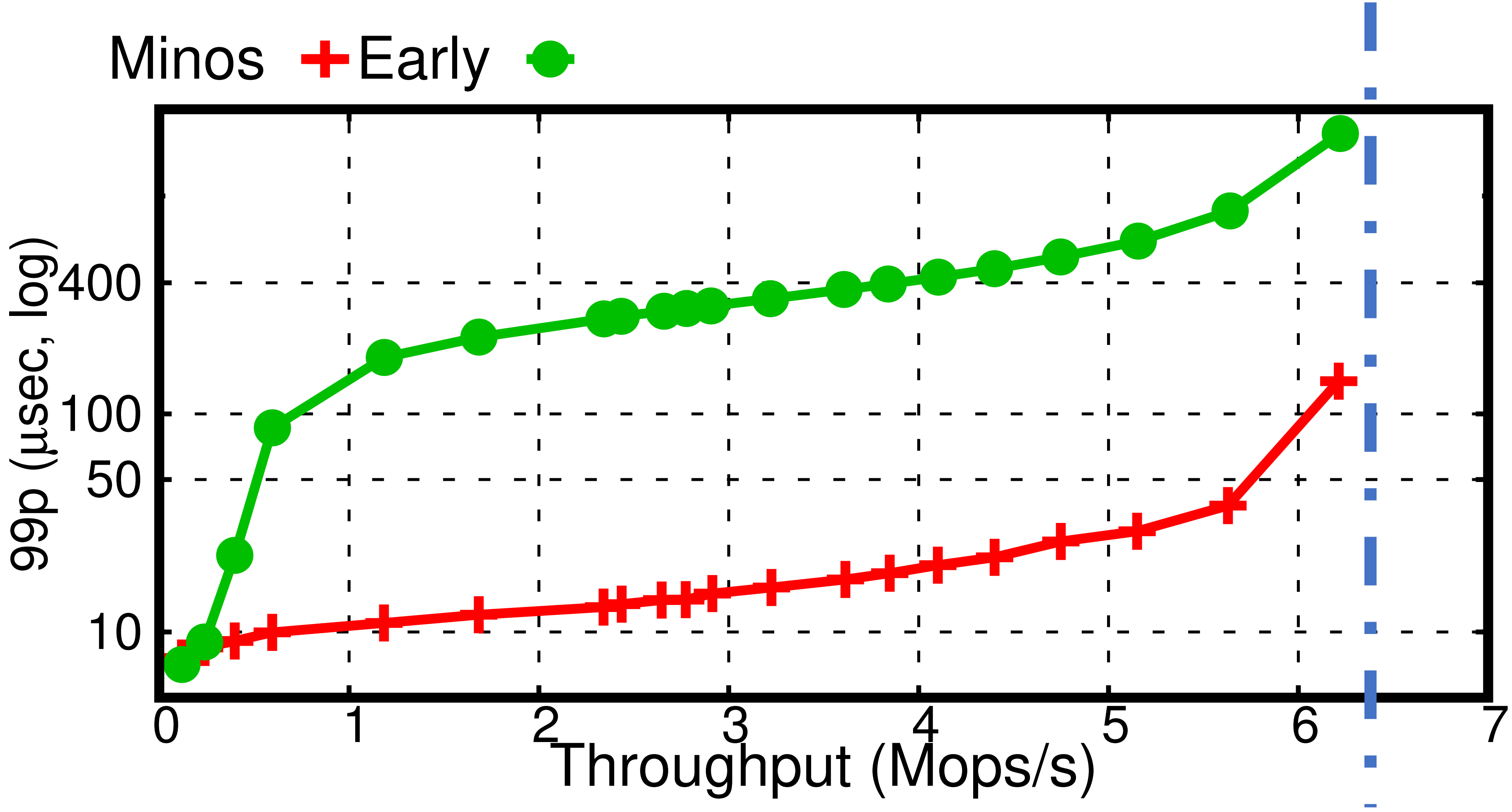
Why? No head-of-line blocking



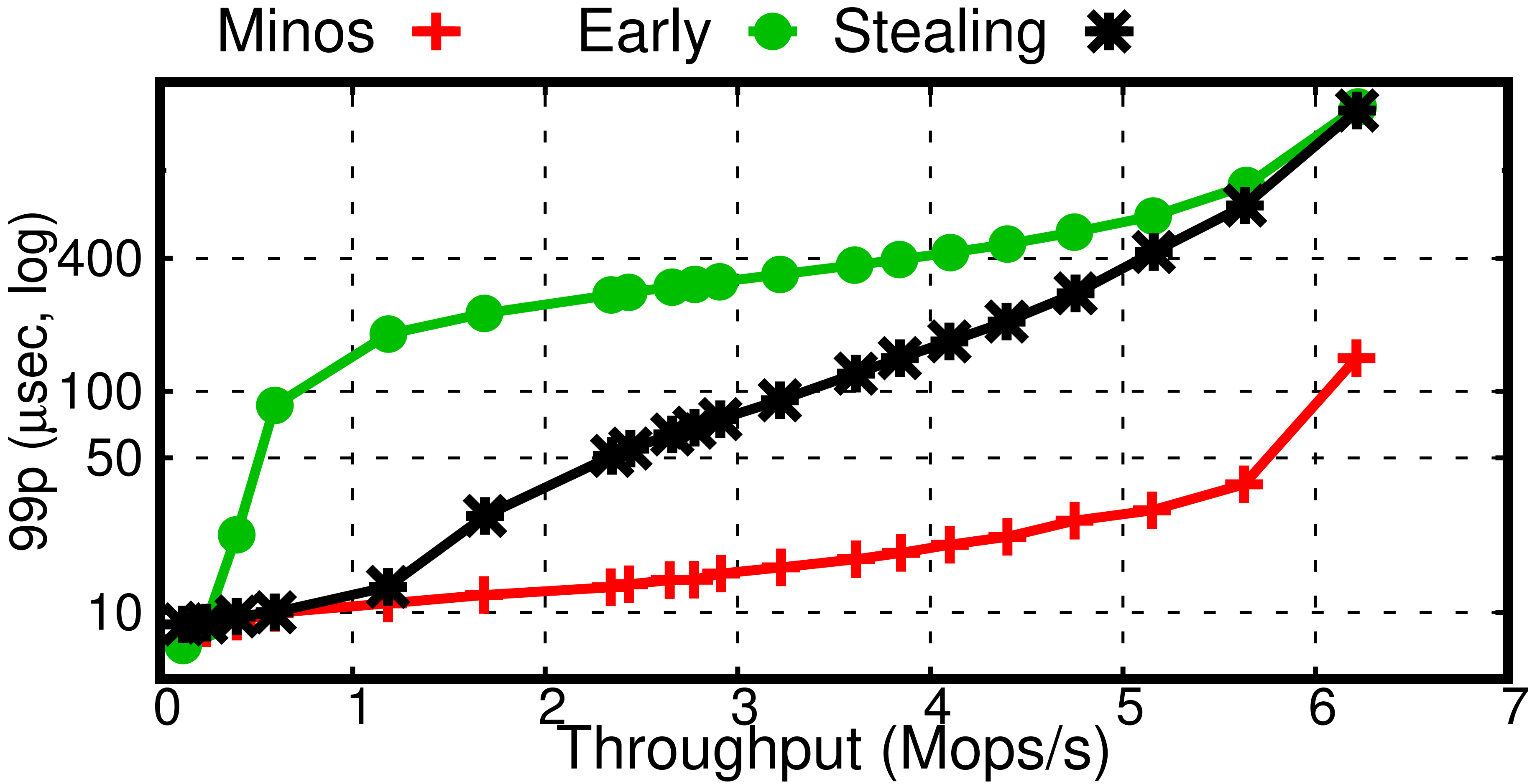
Same maximum throughput



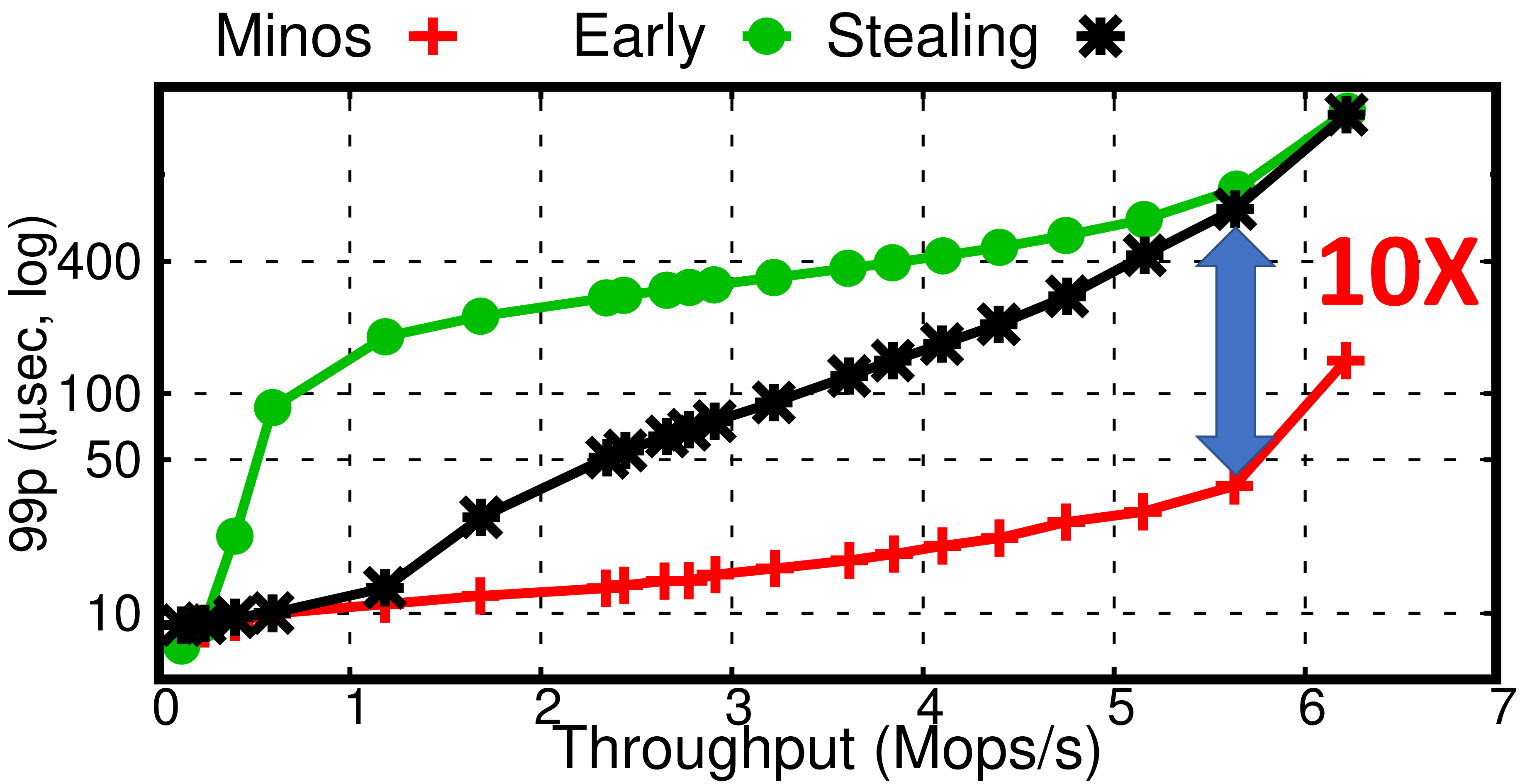
Why? Low dispatch overhead



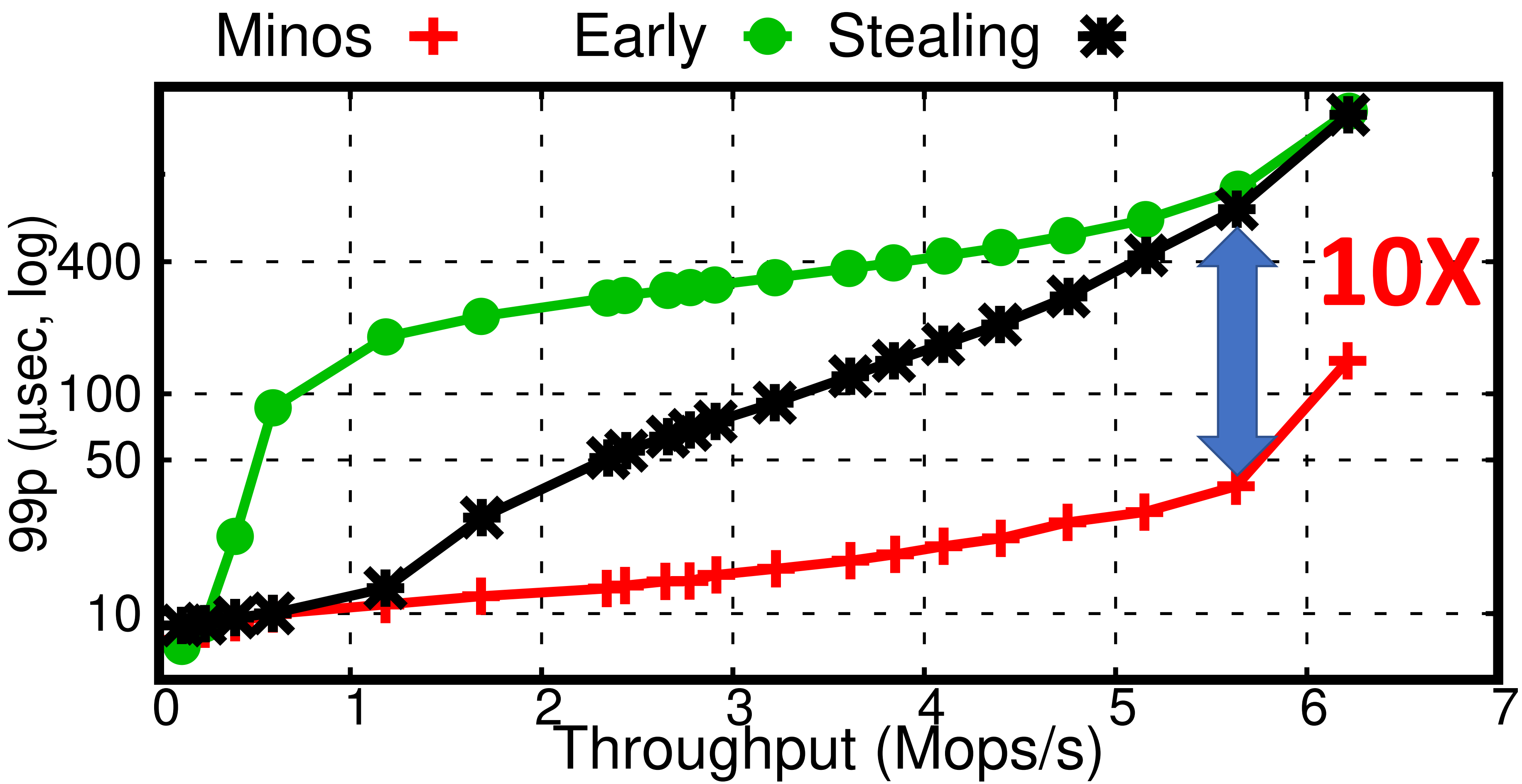
Minos vs stealing



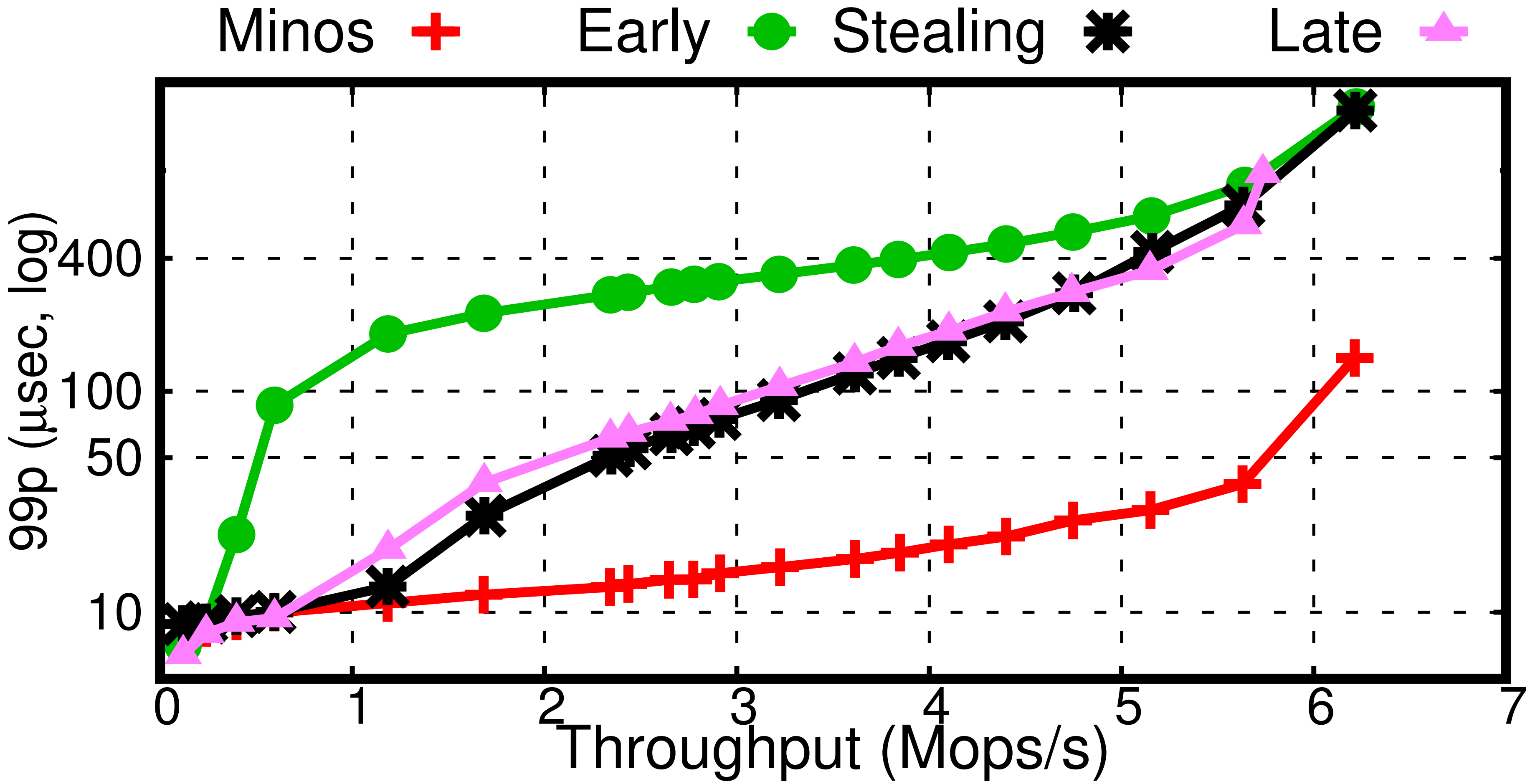
Lower latency



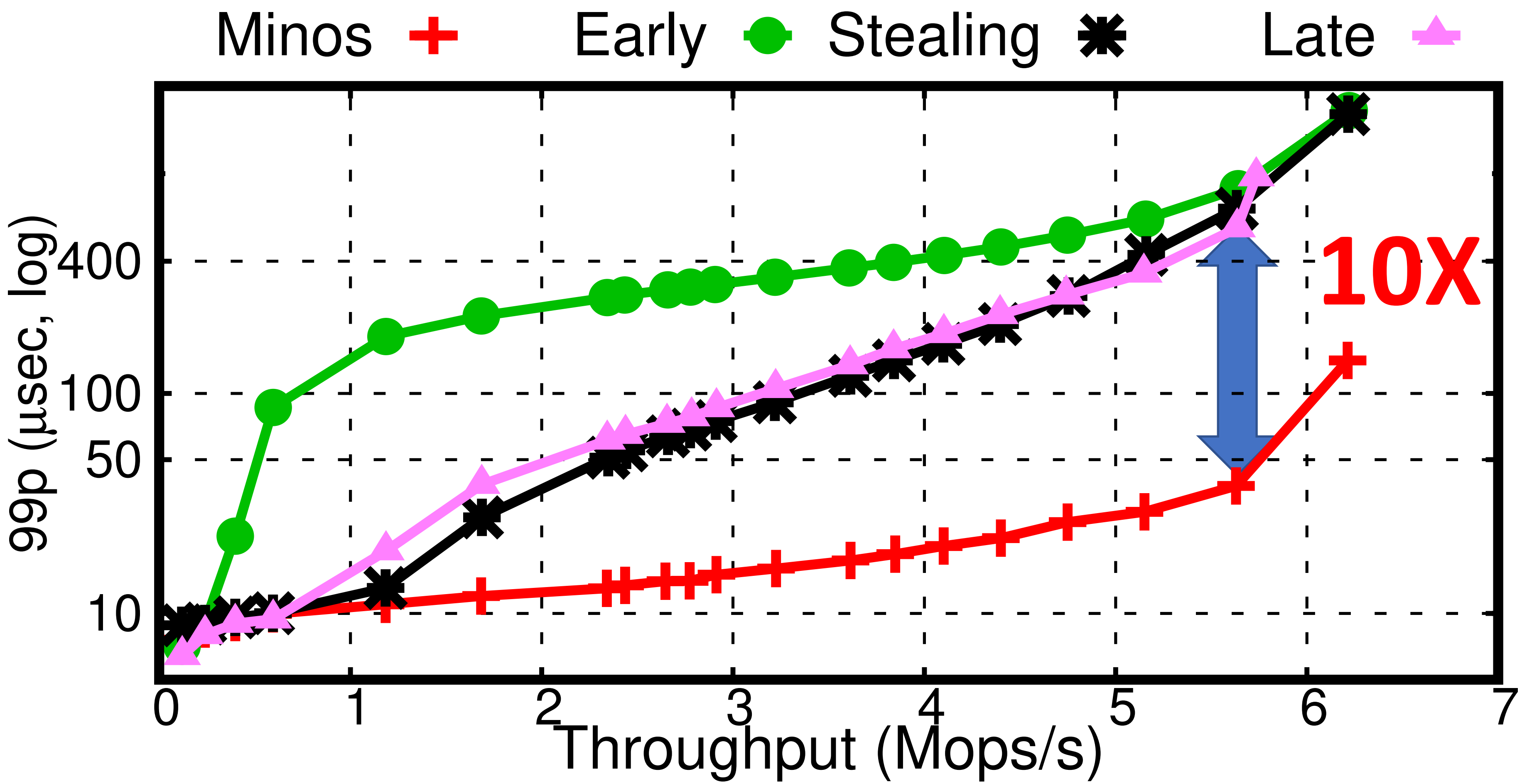
Why? Higher load→ lower stealing



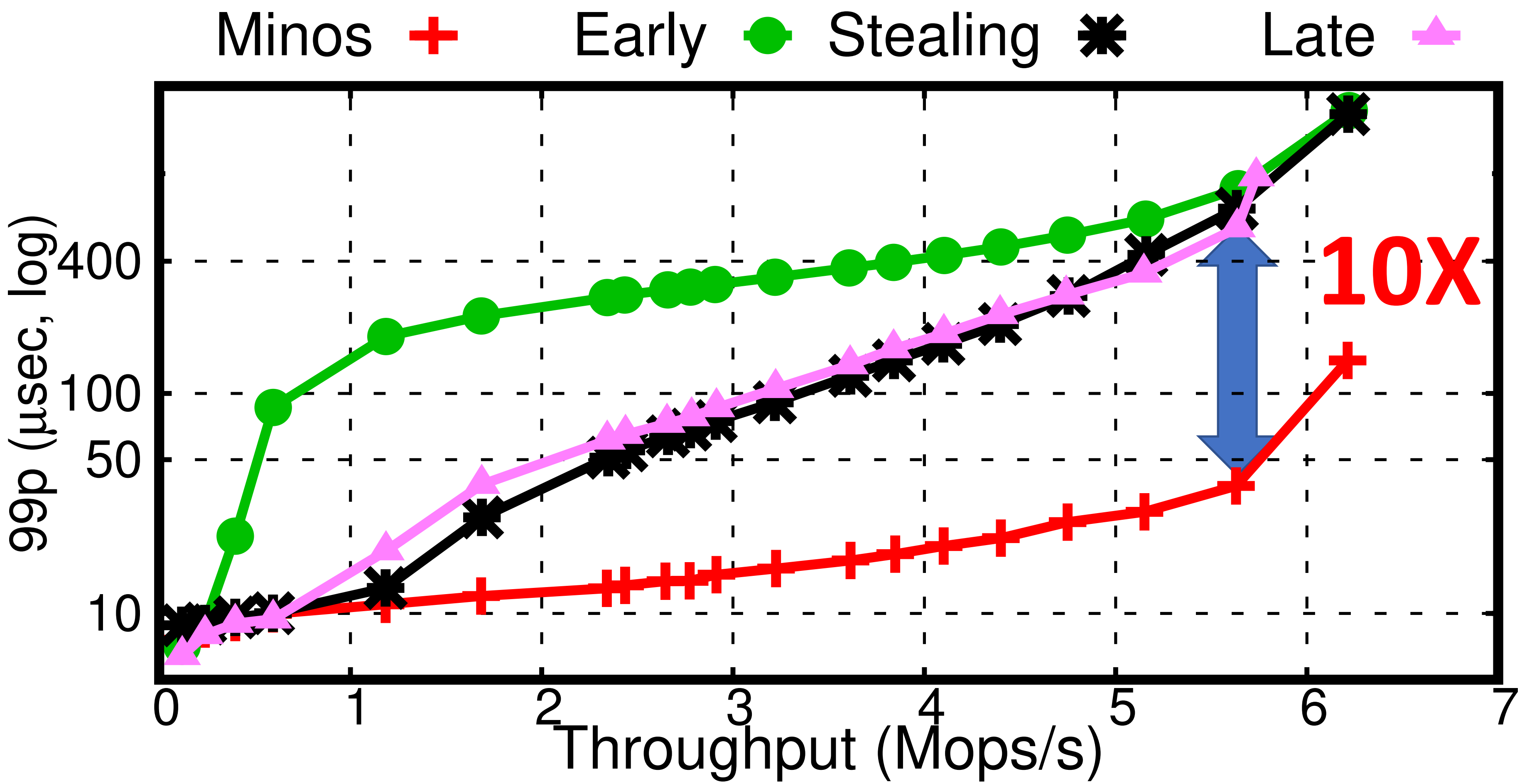
Minos vs late binding



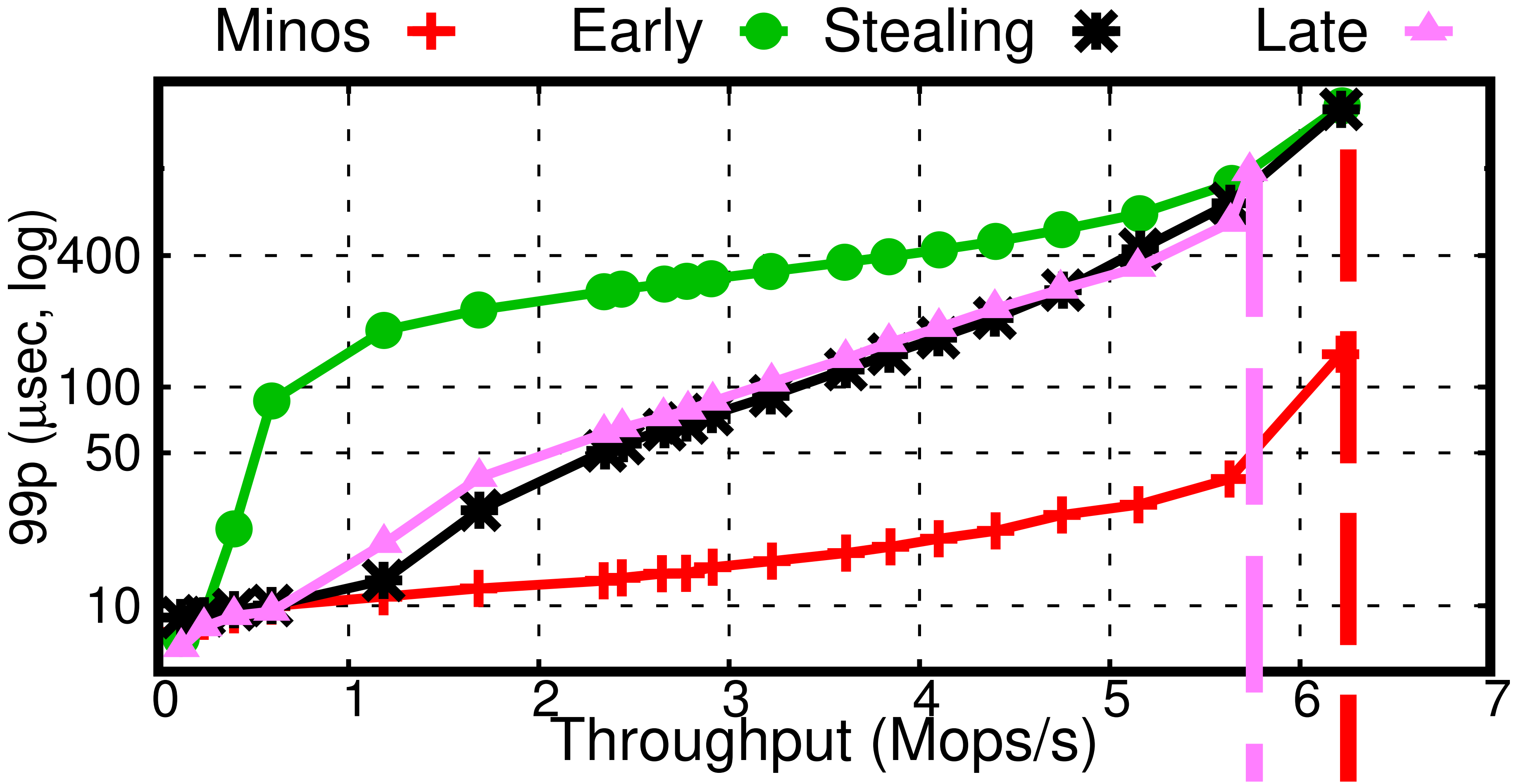
Lower latency



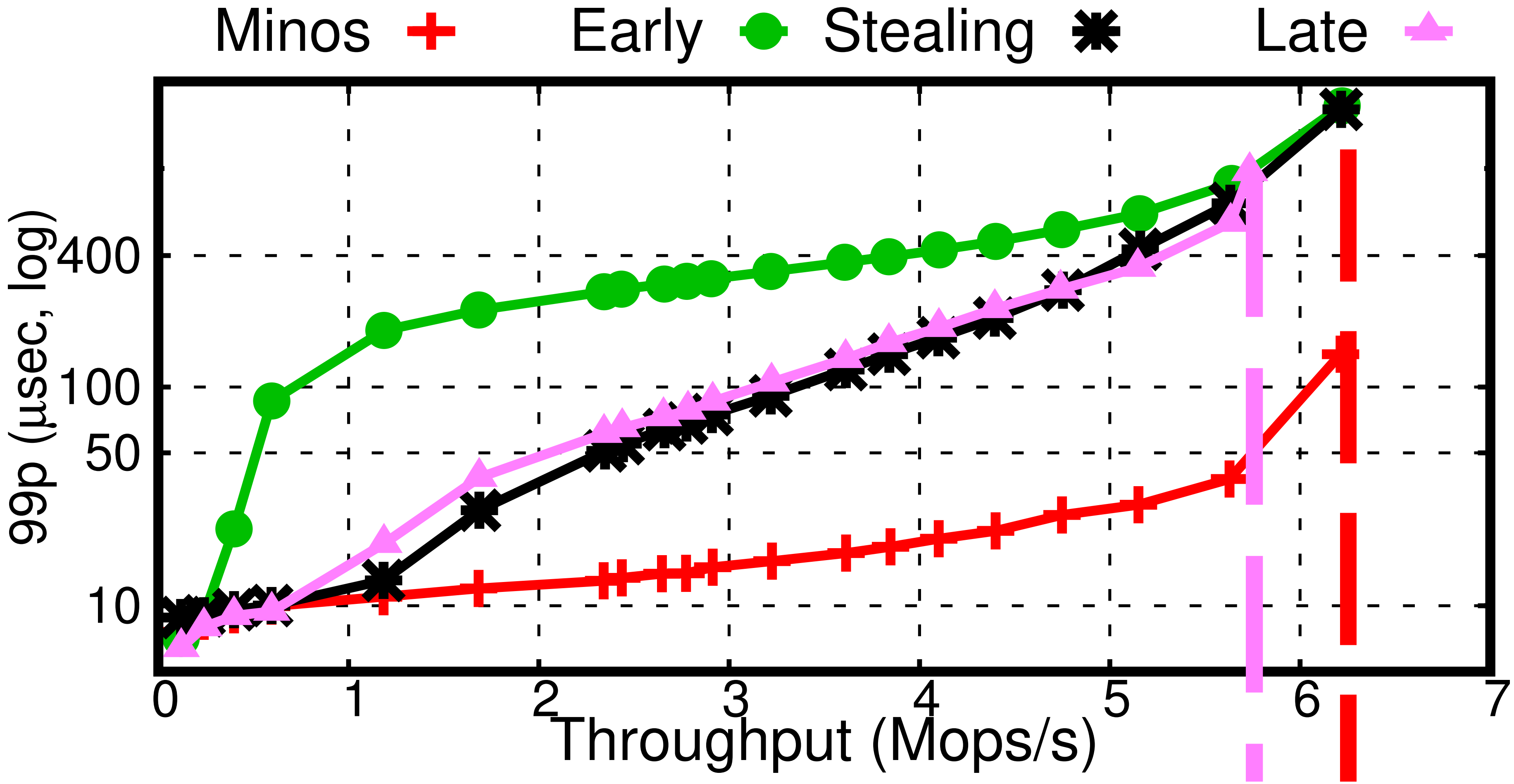
Why? No convoy effect



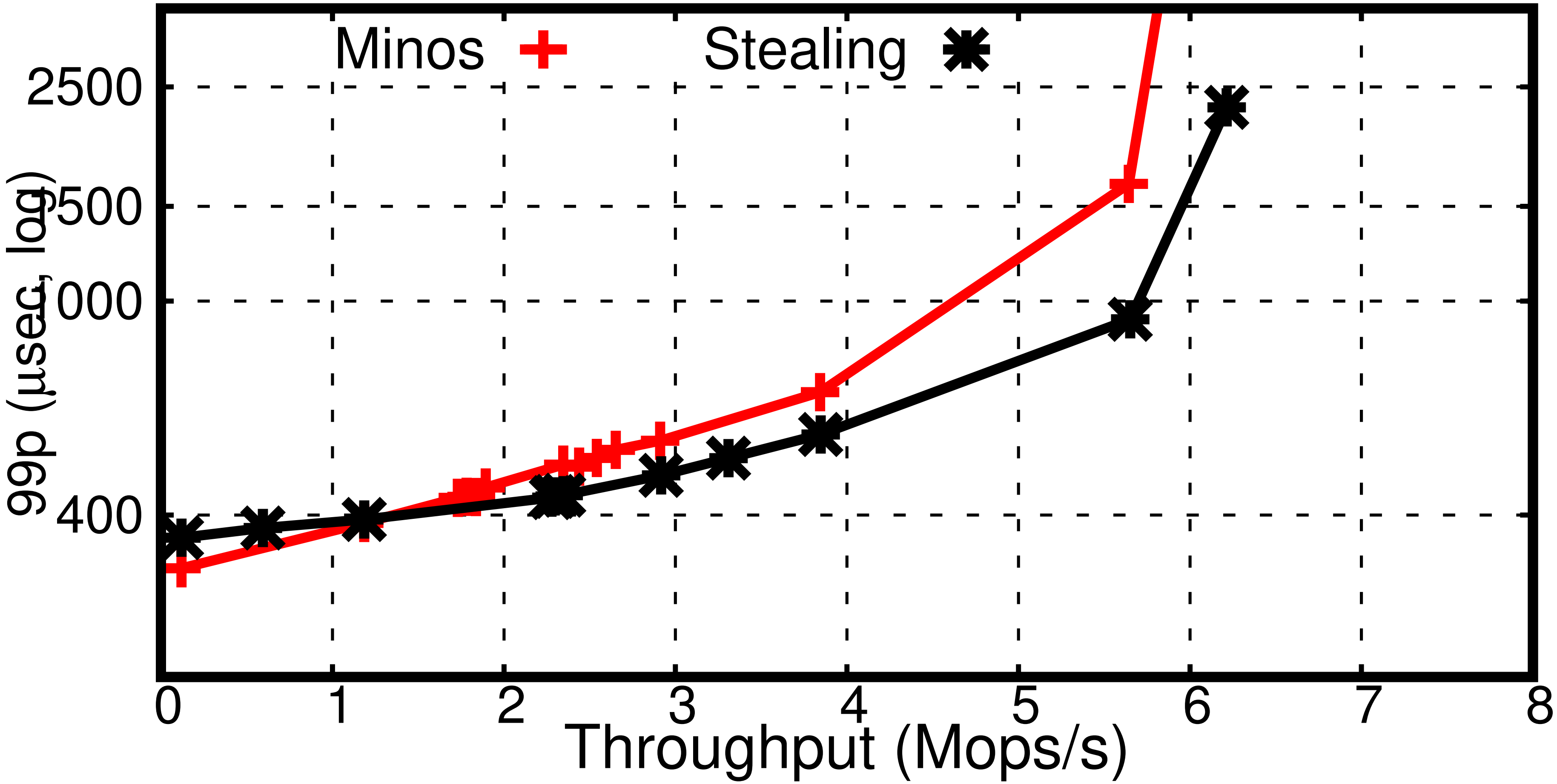
Higher throughput



Why? No dispatch bottleneck



Trade-off: 99th latency of **large** operations



Other results in the paper

- More item size distributions
- Dynamic workload
- Write intensive workload
- Scalability

Conclusion: size-aware sharding

🎯 Improve tail latency in in-memory key-value stores

💡 Serve small and large requests on different cores

- Minos in-memory KV: 10x lower 99th percentile latency

THANK YOU

ANY QUESTIONS?