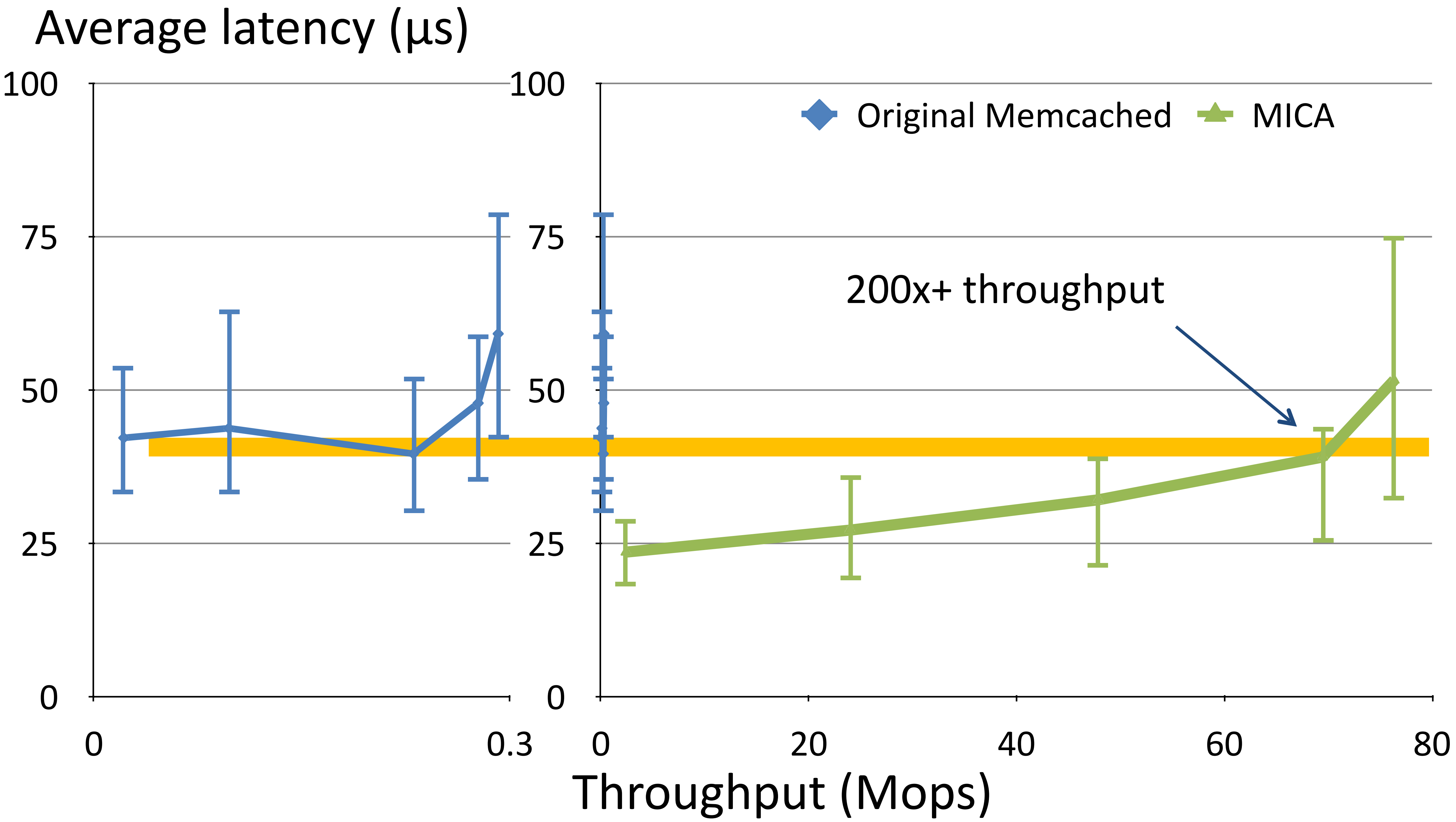
Hardware
Protocols
and Key-Value Storage

David G. Andersen, Michael Kaminsky
and the folks who really did the work:
Hyeontaek Lim, Anuj Kalia, Dong Zhou

Throughput-Latency on Ethernet



Original Memcached using standard socket I/O; both use UDP

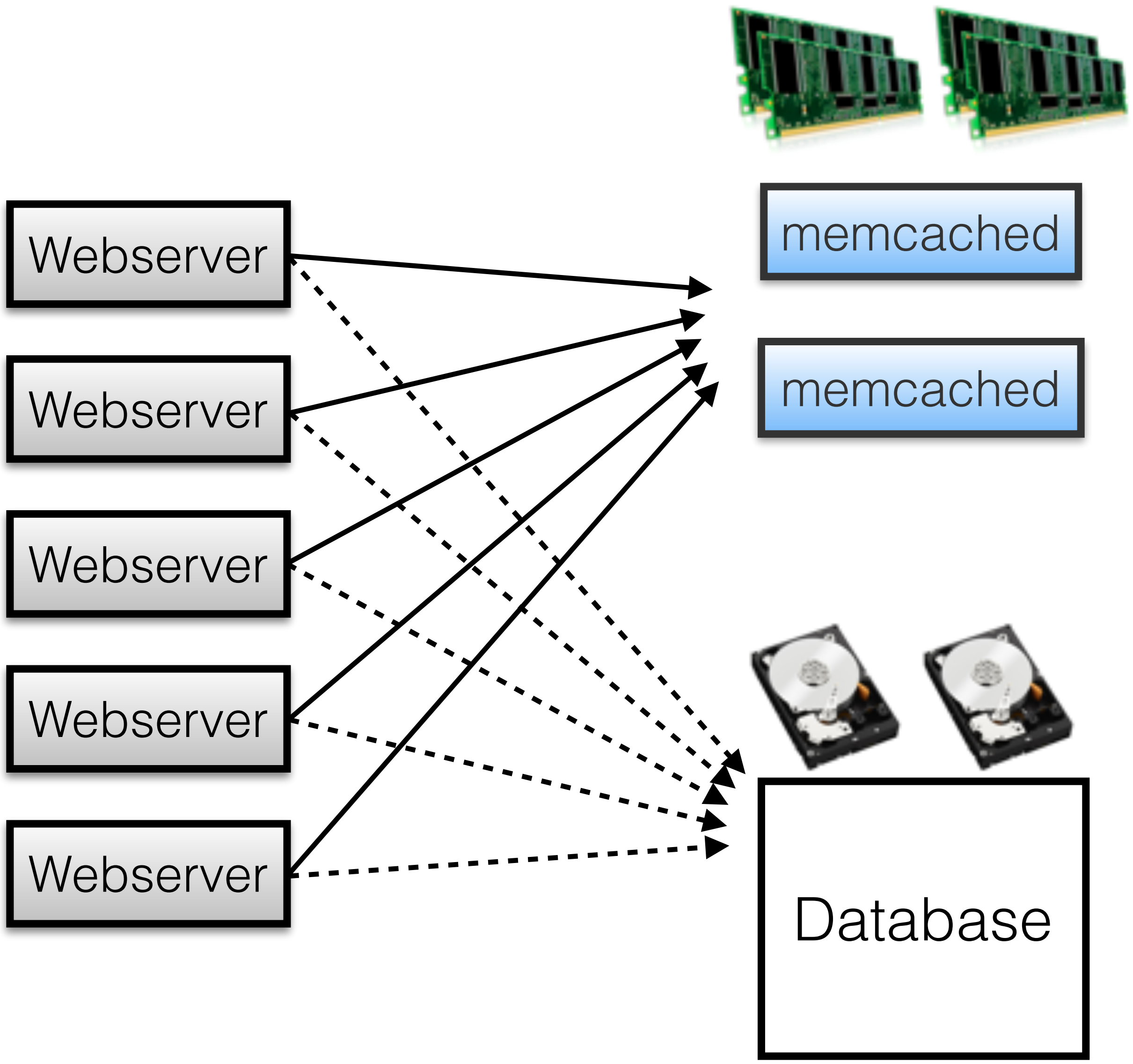
Computational Efficiency

Memory Efficiency

Algorithmic Optimization

Architectural Tailoring

In-memory KV stores



Webserver

Webserver

Webserver

Webserver

Webserver

memcached

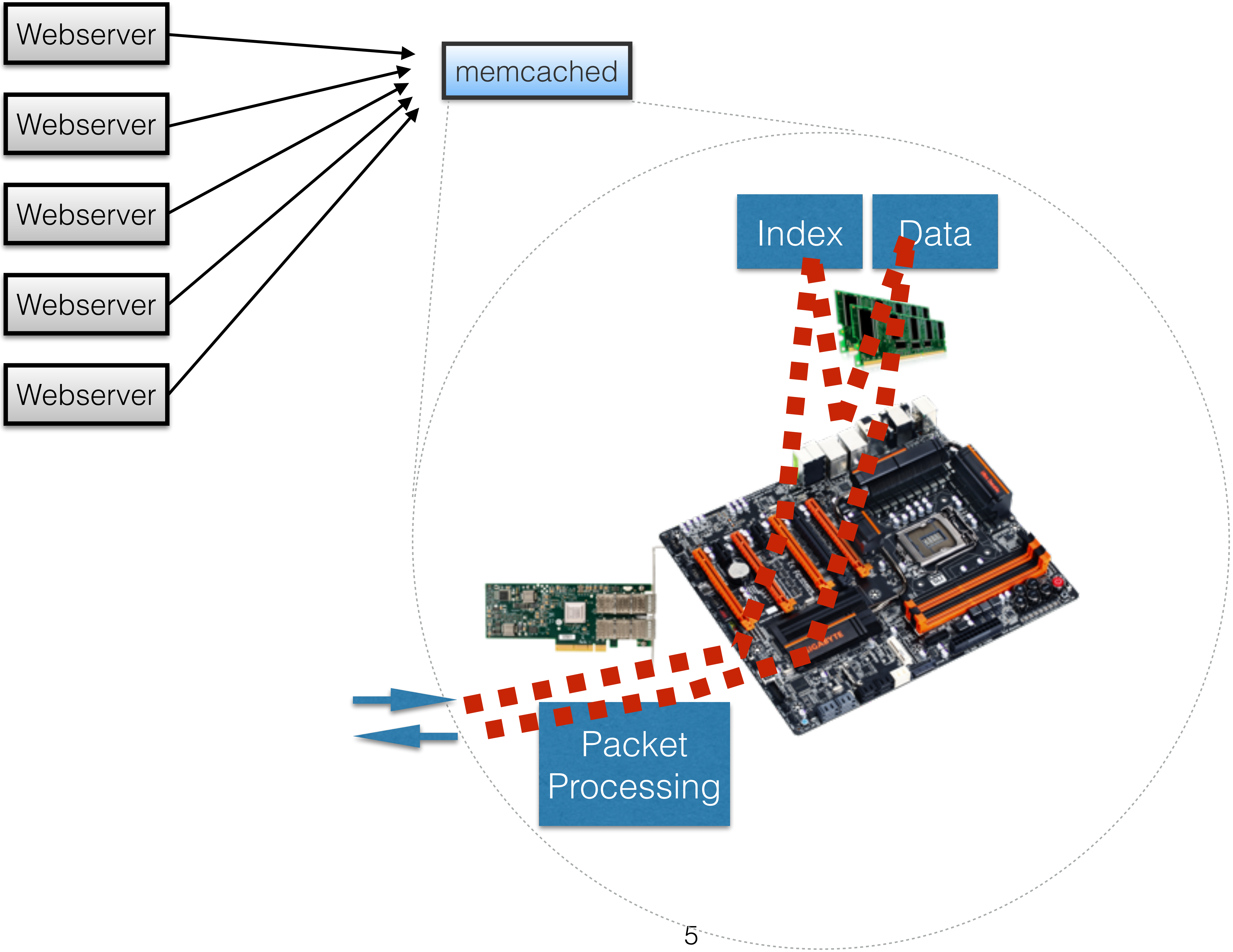
memcached

Database

Interface: GET, PUT

Requirements:

- Low latency
- High request rate



Webserver

Webserver

Webserver

Webserver

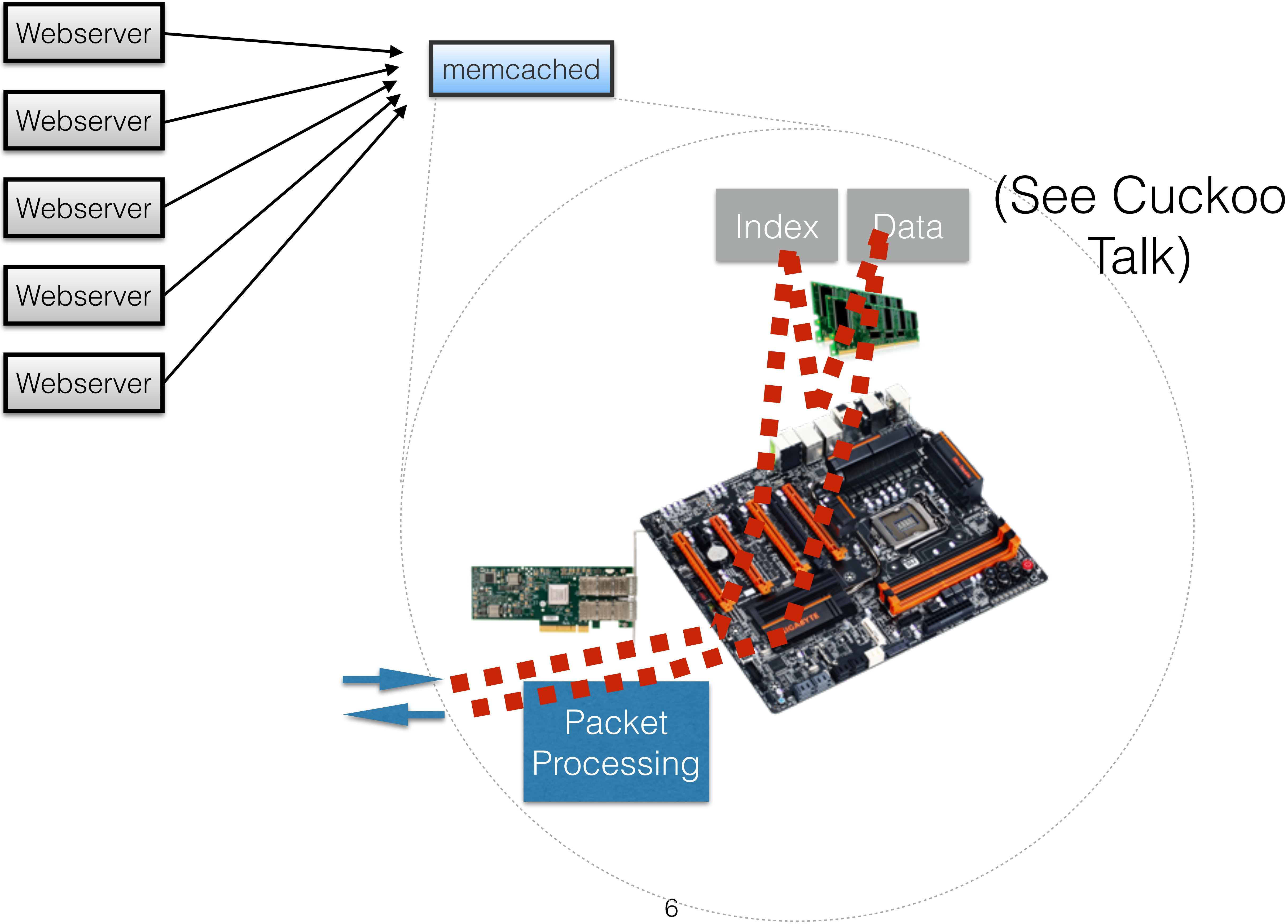
Webserver

memcached

Index

Data

Packet Processing



Webserver

Webserver

Webserver

Webserver

Webserver

memcached

Index

Data

Packet Processing

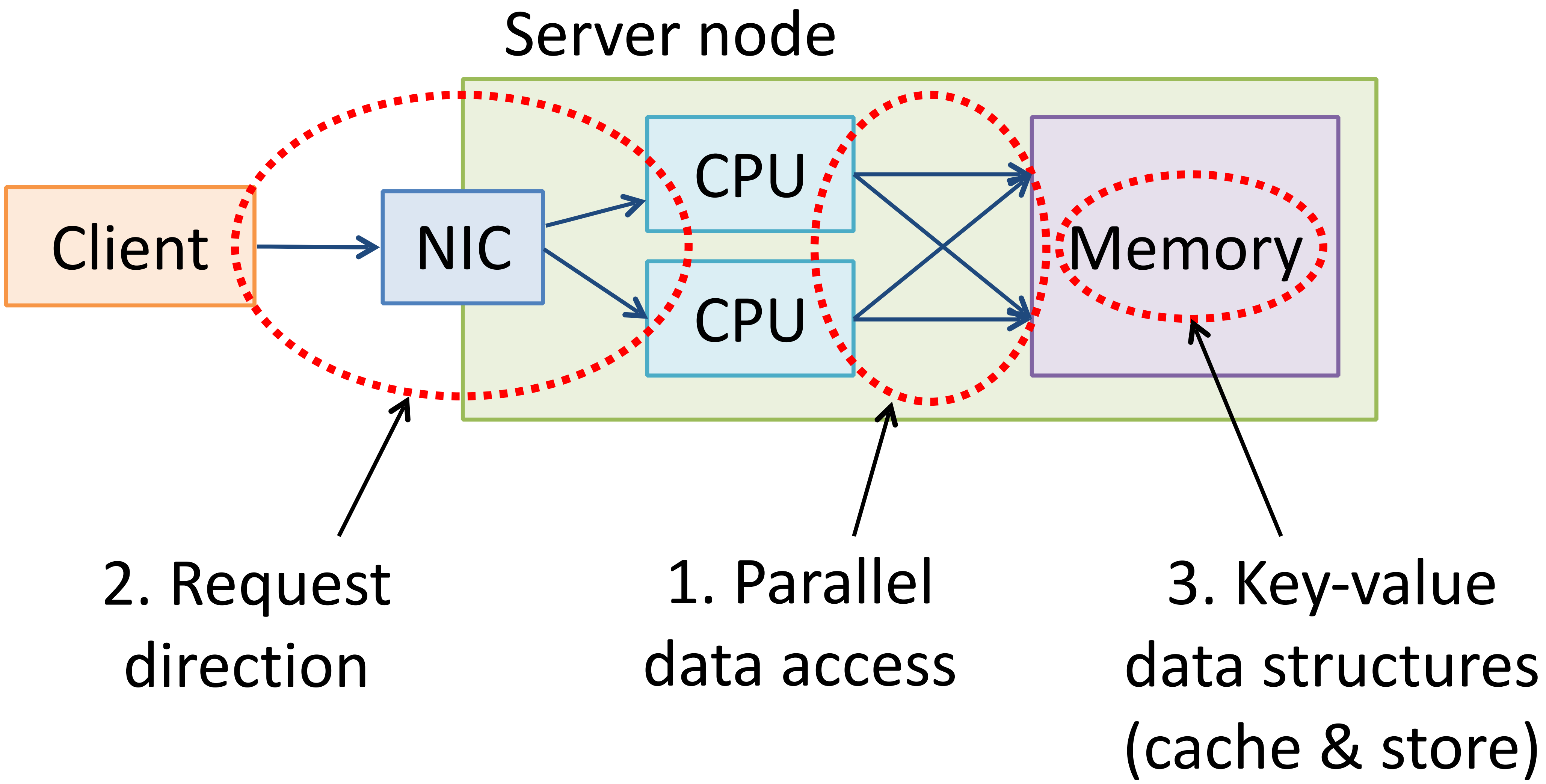
(See Cuckoo Talk)

- How to get requests (packets) in and out?
- How to design & implement the index and datastore?
- In ways that work with *modern* hardware
  - Multicore, NUMA, 40gbps NICs, etc.

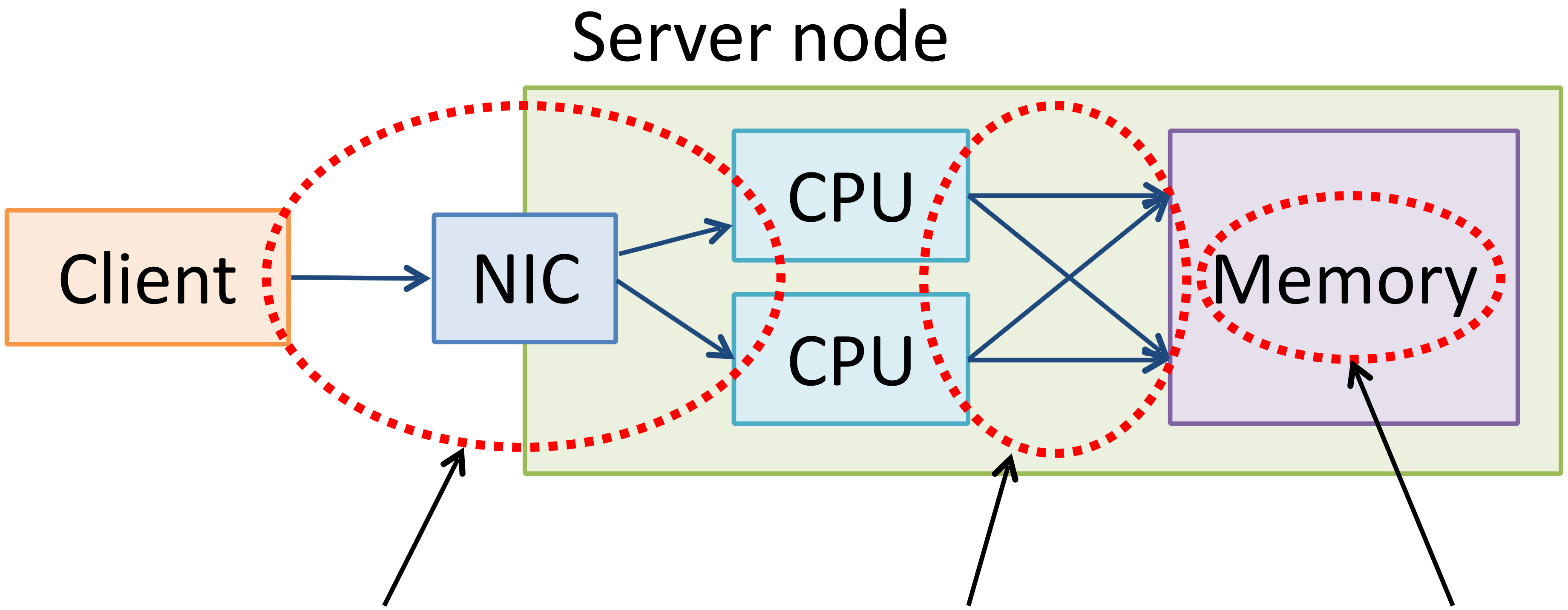
MICA **[NSDI’14]**	**HERD** **[SIGCOMM’14]**
Ethernet	Infiniband / RoCE
Intel DPDK	RDMA

MICA Approach

- **MICA**: Redesigning in-memory key-value storage
  - Applies new SW architecture and data structures to general-purpose HW in a **holistic** way



Parallel Data Access



2. Request direction

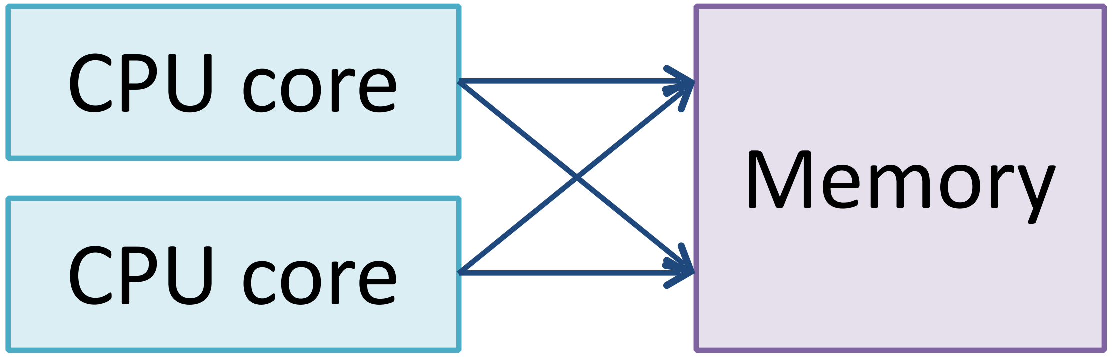
1. Parallel data access

3. Key-value data structures

- Modern CPUs have many cores (8, 15, …)
- How to exploit CPU parallelism efficiently?

Parallel Data Access Schemes

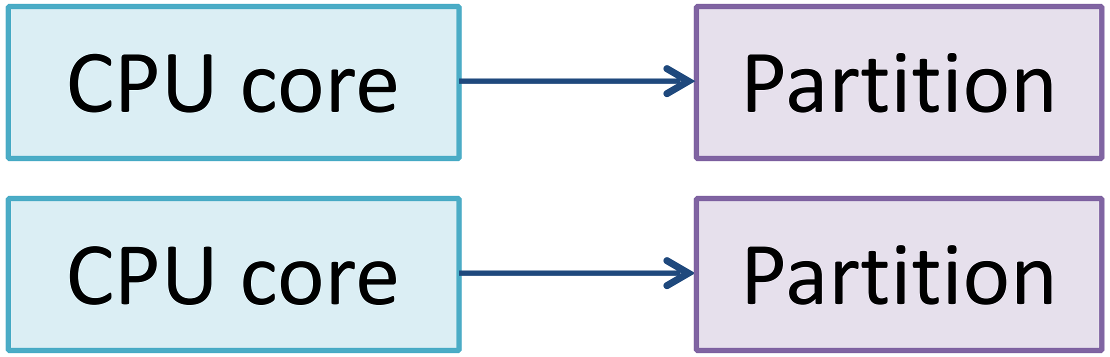
Concurrent Read Concurrent Write



+ Good load distribution

- Limited CPU scalability
  (e.g., synchronization)
- Cross-NUMA latency

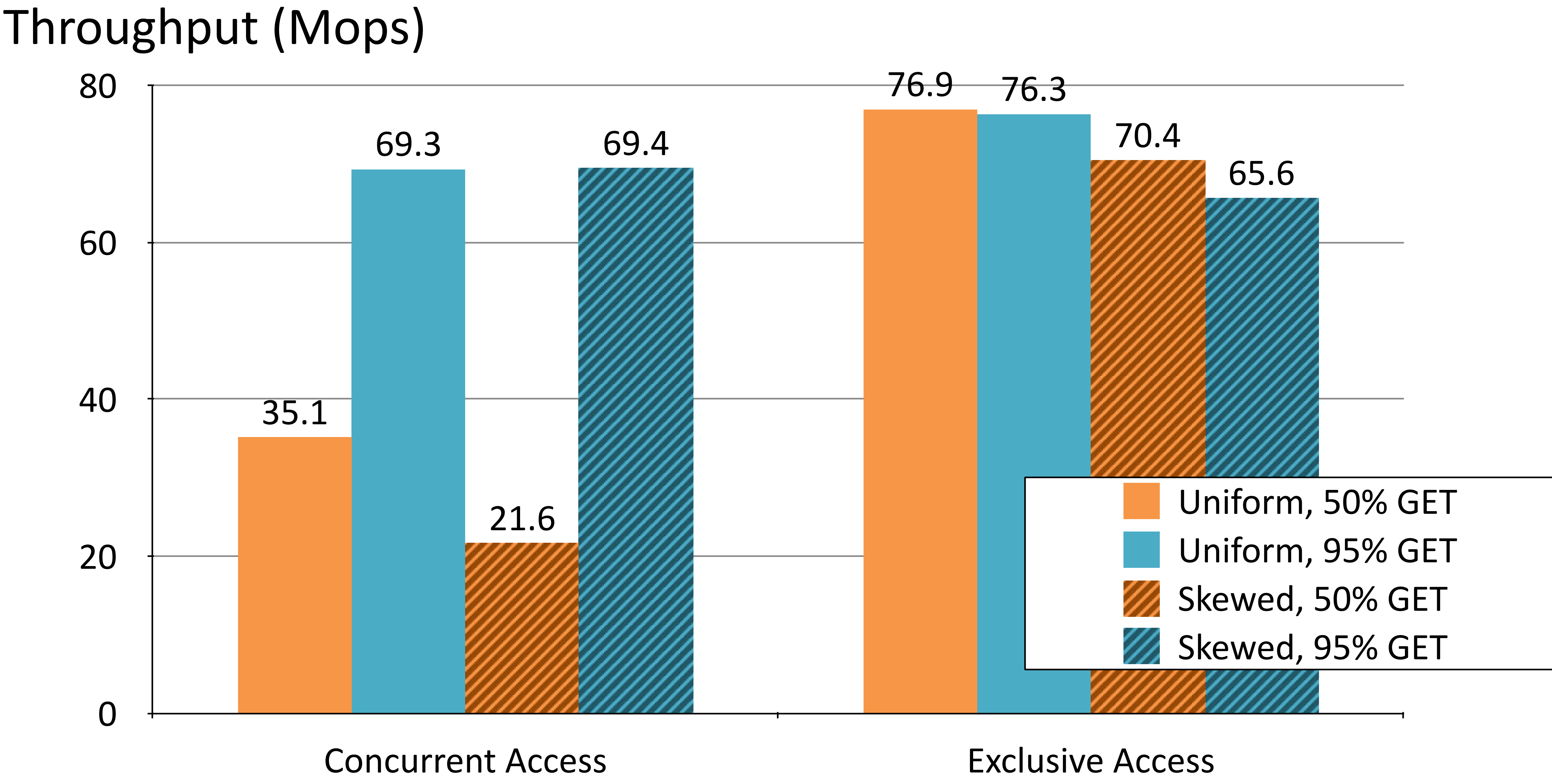
Exclusive Read
Exclusive Write



+ Good CPU scalability

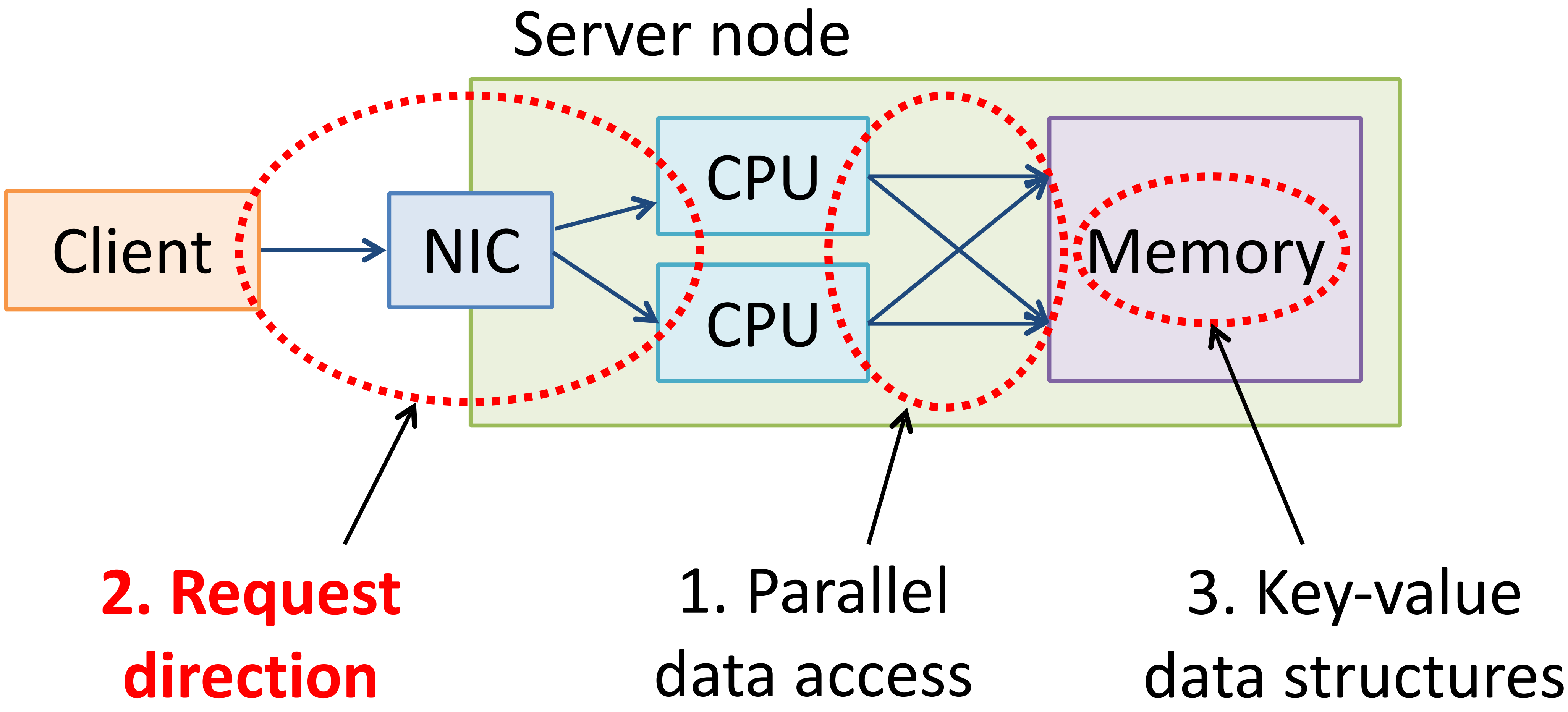
- *Potentially* low performance under skewed workloads

In MICA, Exclusive Outperforms Concurrent



End-to-end performance with kernel bypass I/O

Request Direction



2. Request direction

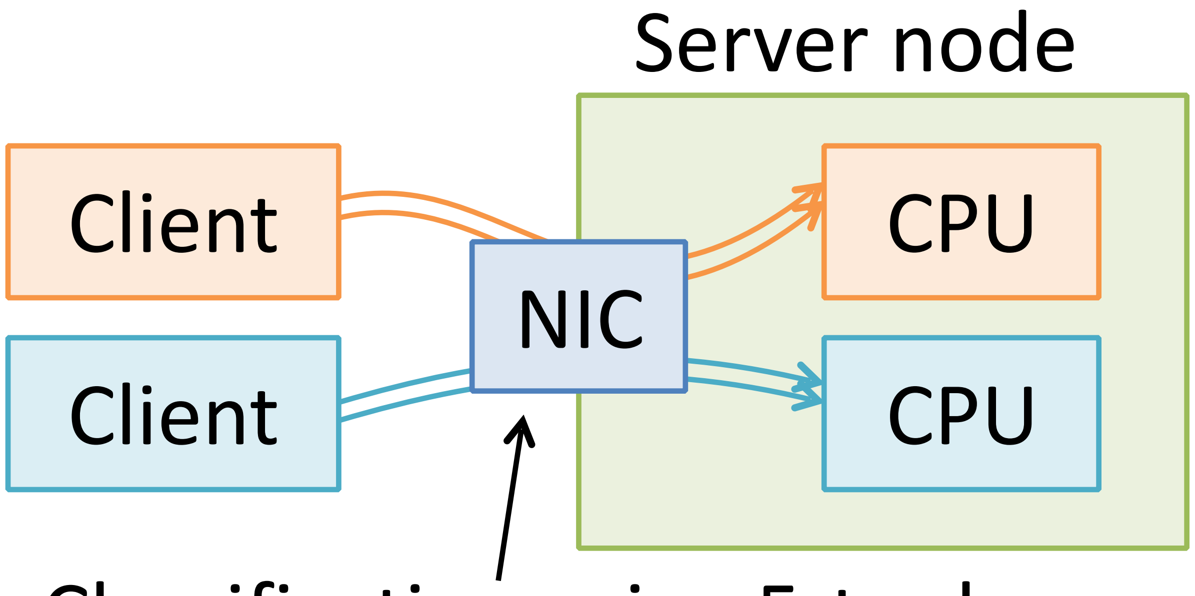
1. Parallel data access

3. Key-value data structures

- Sending requests to appropriate CPU cores for better data access locality
- Exclusive access benefits from correct delivery
  - Each request must be sent to corresp. partition's core

Request Direction Schemes

Flow-based Affinity

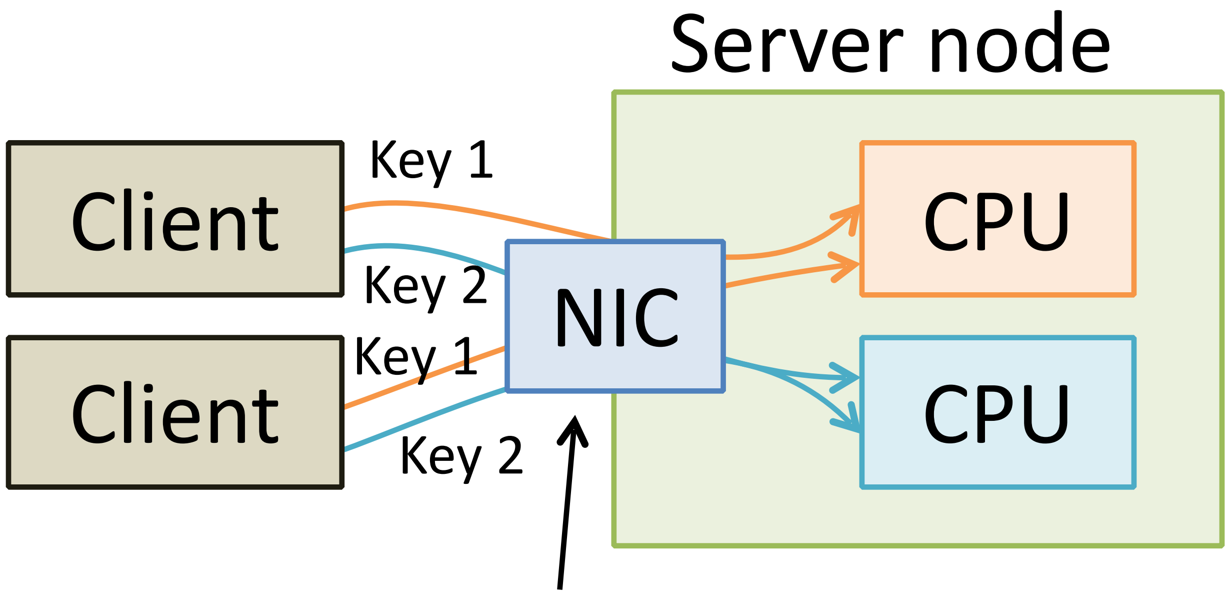


Classification using 5-tuple

+ Good locality for flows (e.g., HTTP over TCP)

- Suboptimal for small key-value processing

Object-based Affinity

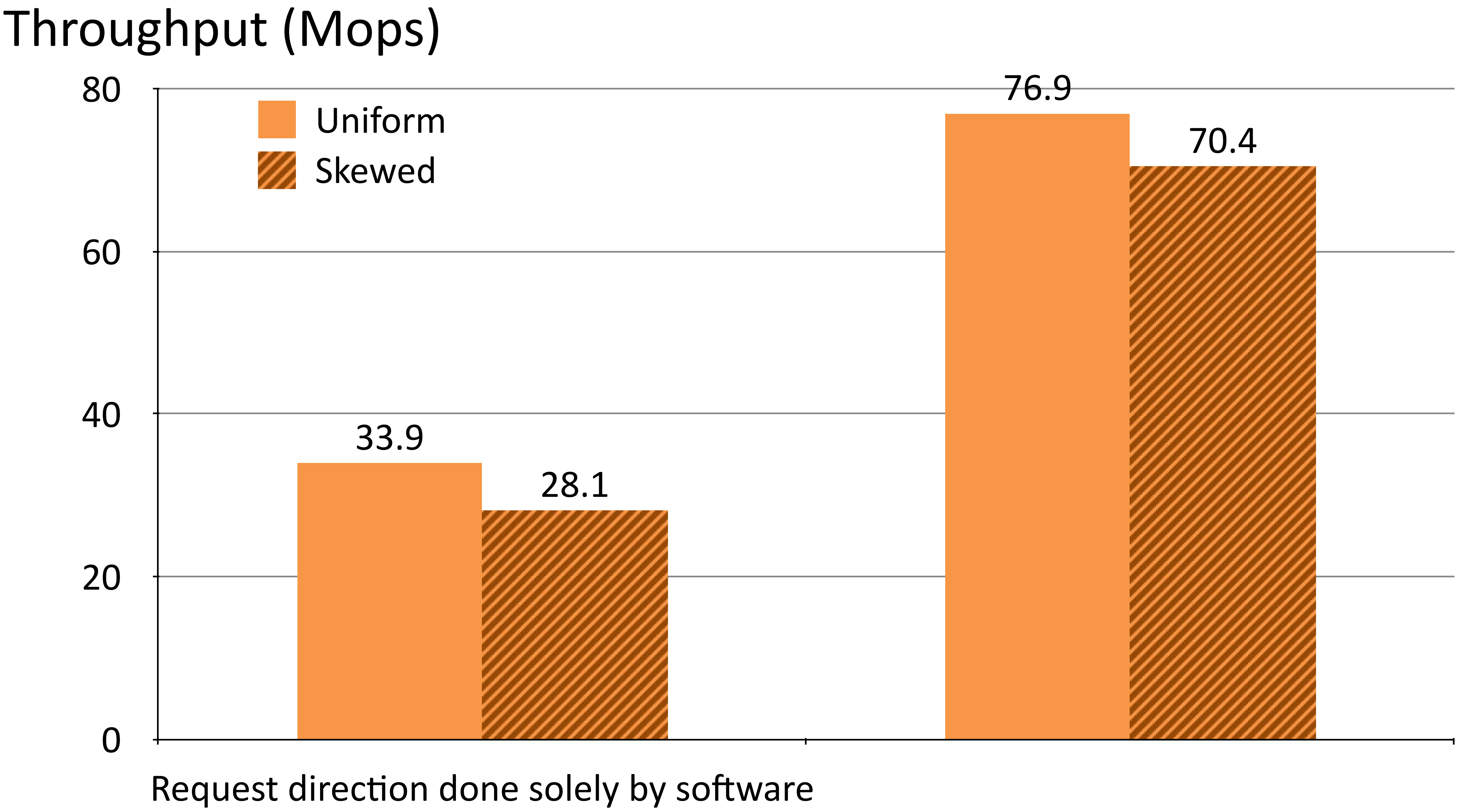


Classification depends on request content

+ Good locality for key access

- **Client assist** or special HW support needed for efficiency

Crucial to Use NIC HW for Request Direction



Using exclusive access for parallel data access

Plus some cool data structures inside

(see Lim et al., NSDI 2014)

Result:
The fastest network-based key-value server that we know of.

2 socket Xeon server can nearly saturate 80Gbps of Ethernet (8x10Gbps).

Protocol changes to let NICs direct requests to the right core

Careful attention to NUMA and locality

OS & Stack bypass to eliminate overhead

RDMA

Remote Direct Memory Access:
A network feature that allows direct access to the memory of a remote computer.

HERD

1. Improved understanding of RDMA through micro-benchmarking

2. High-performance key-value system:
   - Throughput: 26 Mops *(2X higher than others)*
   - Latency: 5 μs *(2X lower than others)*

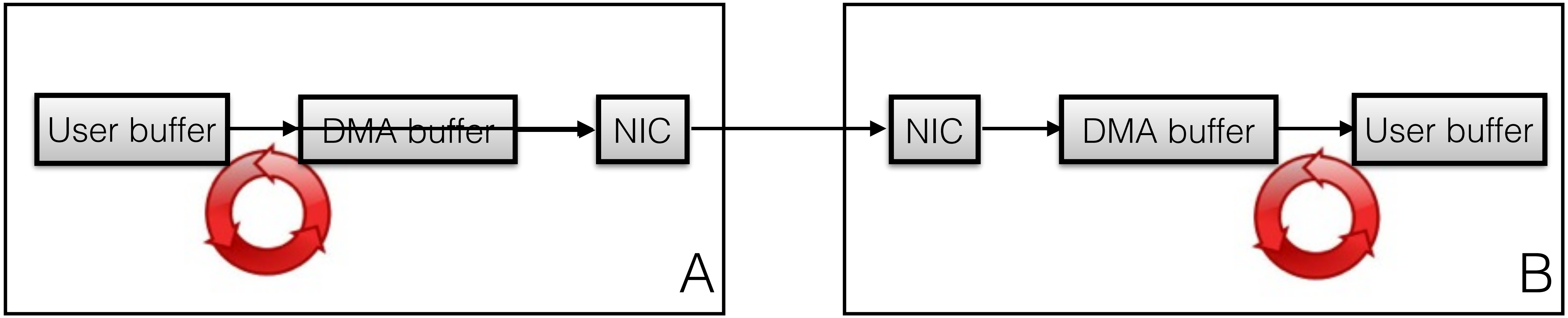
RDMA intro

Features:

- Ultra-low latency: 1 µs RTT
- Zero copy + CPU bypass

Providers:

InfiniBand, RoCE,…



User buffer

DMA buffer

NIC

A

NIC

DMA buffer

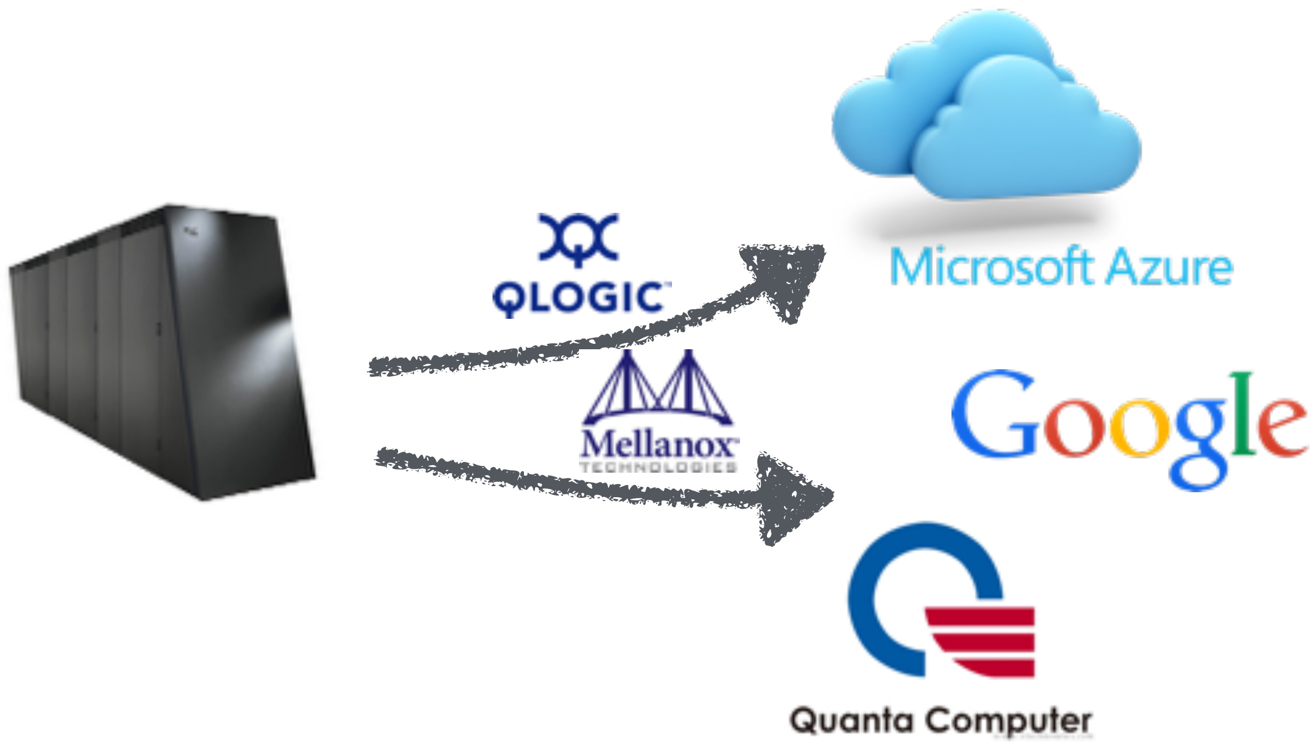
User buffer

B

RDMA in the datacenter

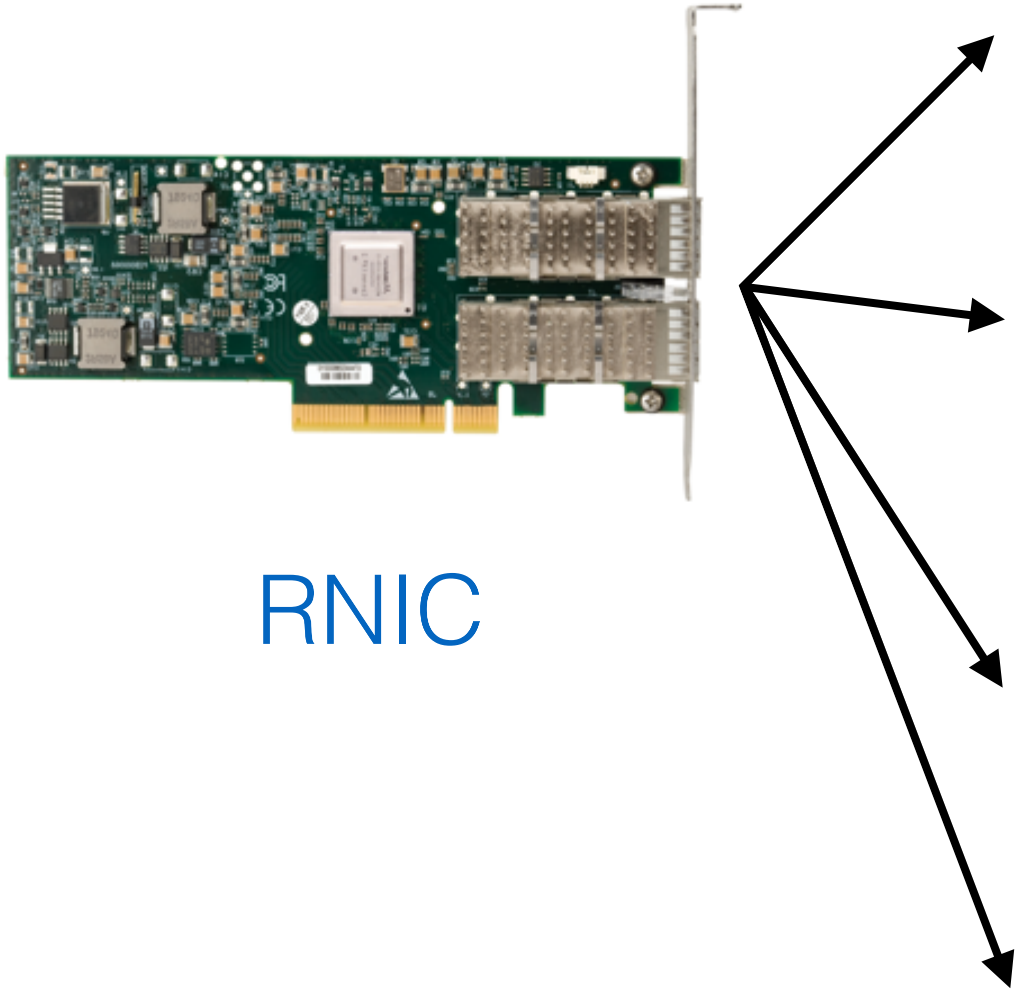
48 port 10 GbE switches

Switch	RDMA	Cost
Mellanox SX1012	**YES**	$5,900
Cisco 5548UP	**NO**	$8,180
Juniper EX5440	**NO**	$7,480



RDMA basics

Verbs



RDMA read:

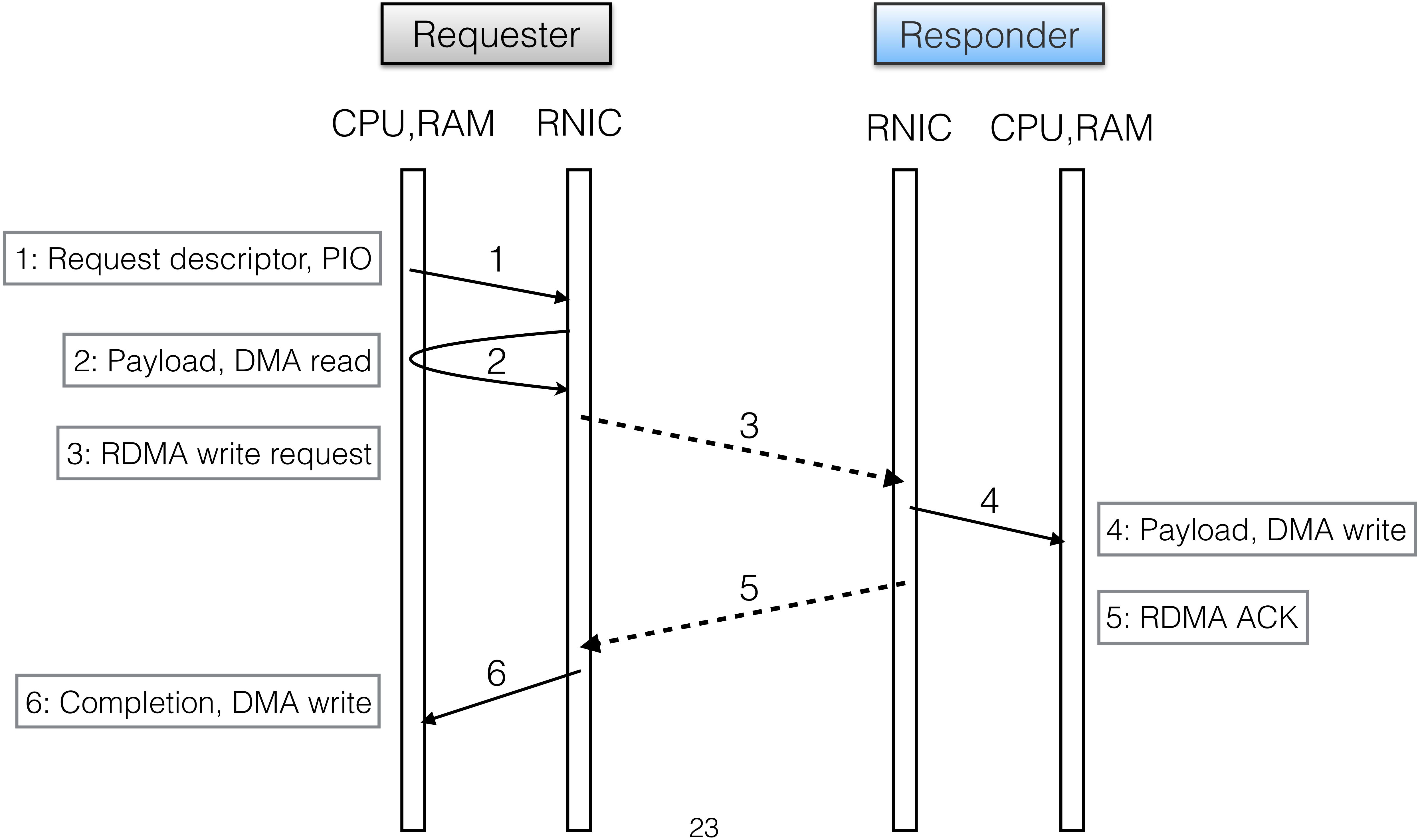
`READ(local_buf, size, remote_addr)`

RDMA write:

`WRITE(local_buf, size, remote_addr)`

⋮

Life of a WRITE



1: Request descriptor, PIO

2: Payload, DMA read

3: RDMA write request

4: Payload, DMA write

5: RDMA ACK

6: Completion, DMA write

Recent systems

Pilaf [ATC 2013]

FaRM-KV [NSDI 2014]: an example usage of FaRM

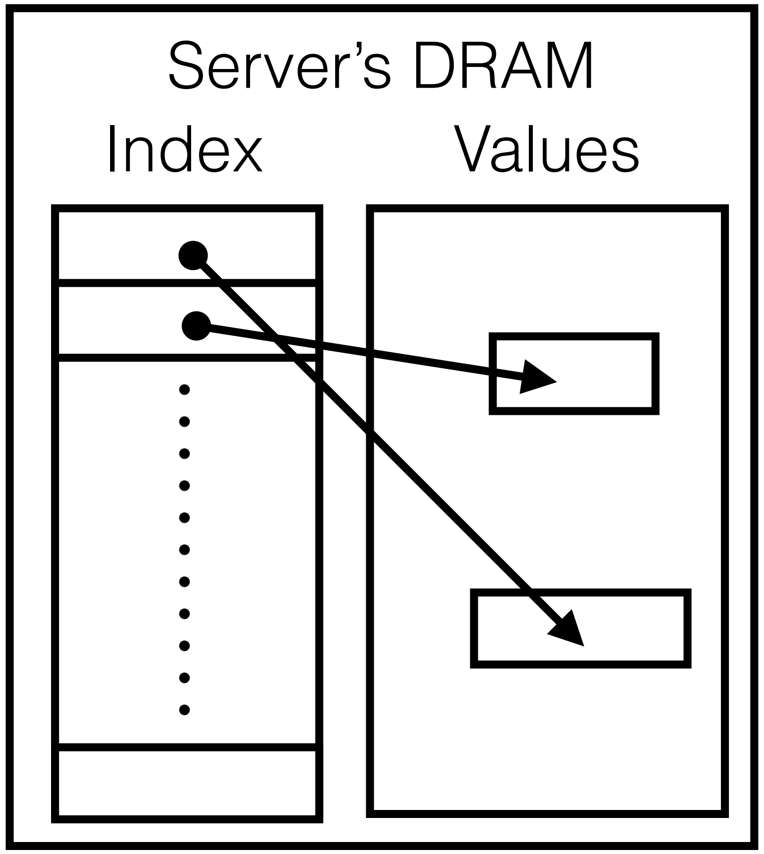
Approach: RDMA reads to access remote data structures

Reason: the allure of CPU bypass

The price of CPU bypass

Key-Value stores have an inherent level of indirection.

An index maps a keys to address. Values are stored separately.



At least 2 RDMA reads required:

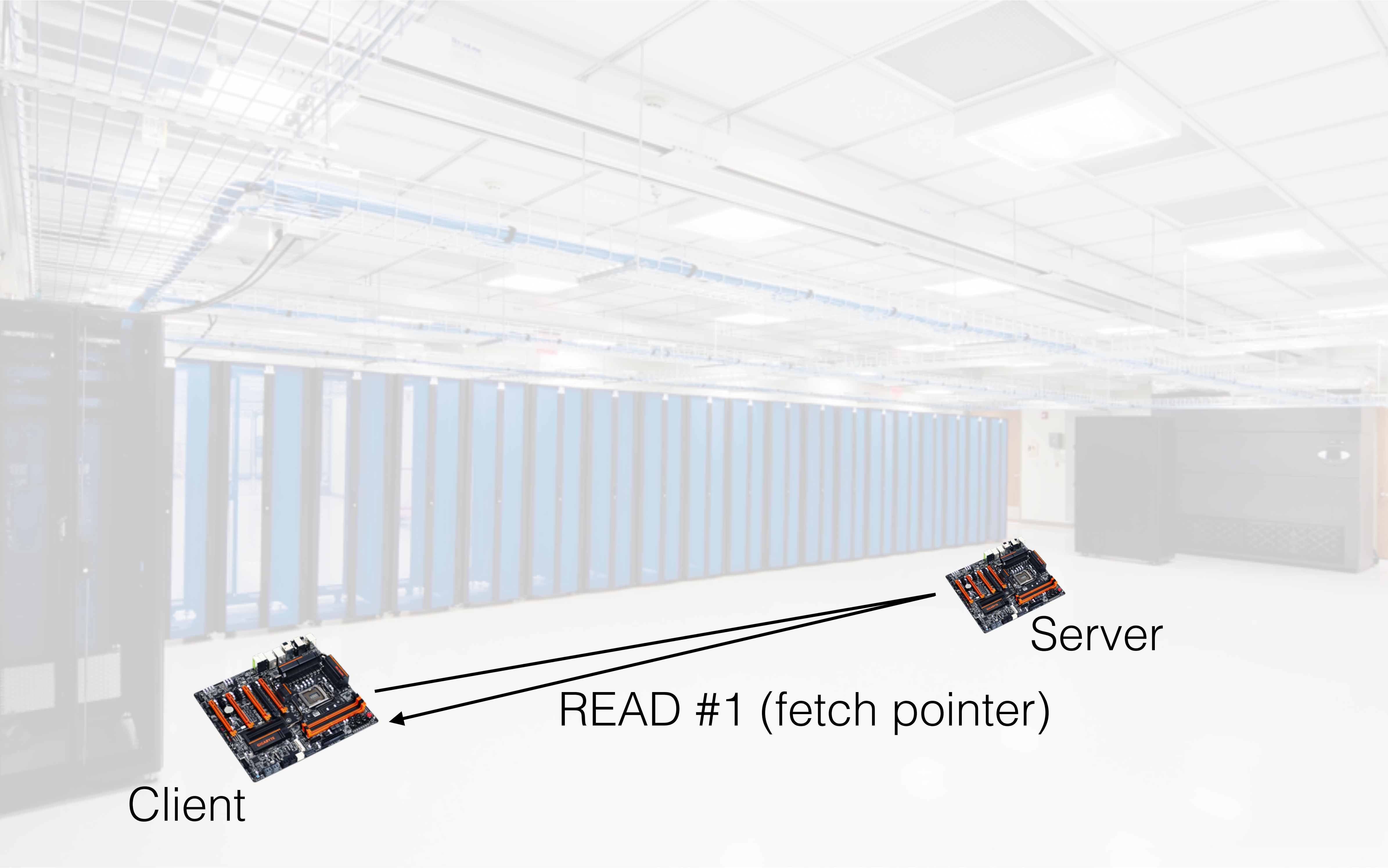
≧ 1 to fetch address

1 to fetch value

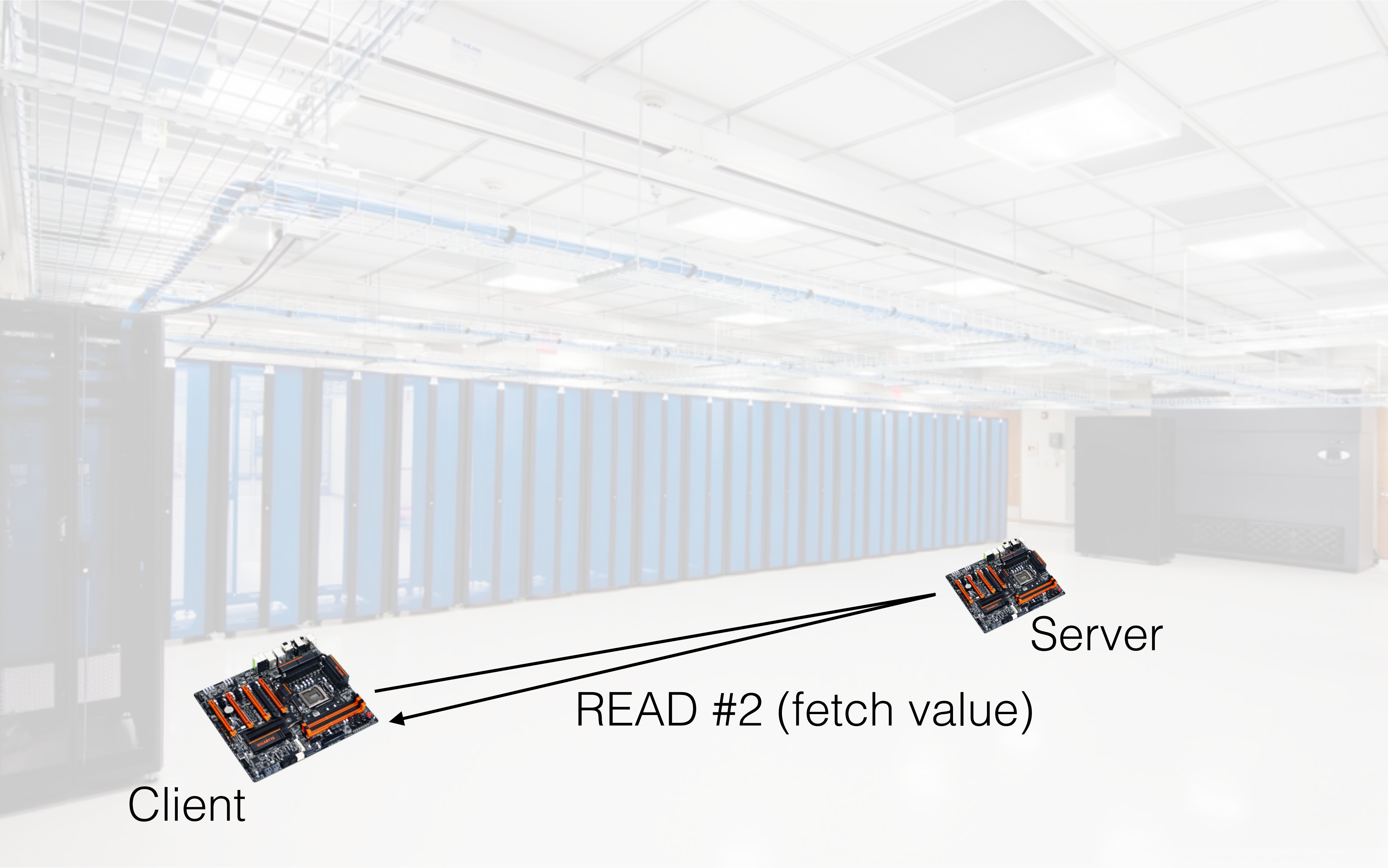
Not true if value is in index

The price of CPU bypass

The price of CPU bypass



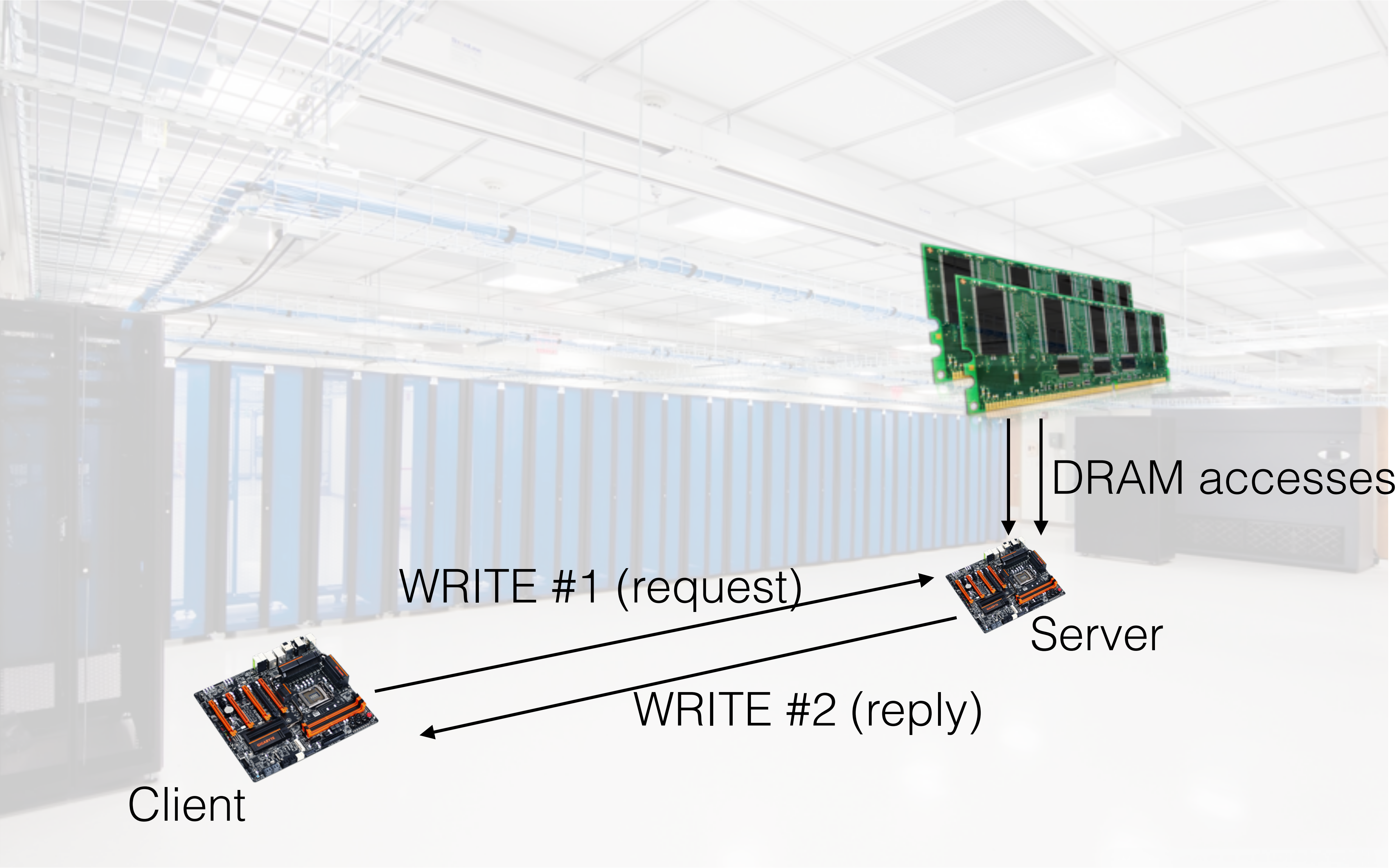
The price of CPU bypass



Our approach

Goal	Main ideas
#1: Use a single round trip	Request-reply with server CPU involvement + WRITEs faster than READs
#2. Increase throughput	Low level verbs optimizations
#3. Improve scalability	Use datagram transport

#1: Use a single round trip



DRAM accesses

WRITE #1 (request)

Server

WRITE #2 (reply)

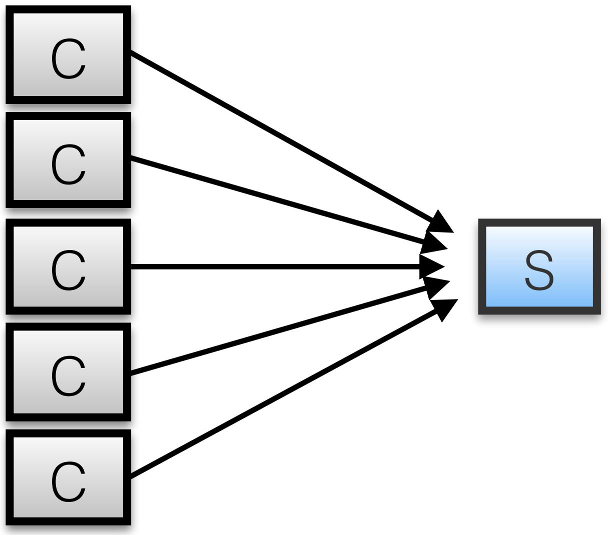
Client

#1: Use a single round trip

Operation	Round Trips	Operations at server’s RNIC
READ-based GET	**2+**	2+ RDMA reads
HERD GET	**1**	2 RDMA writes

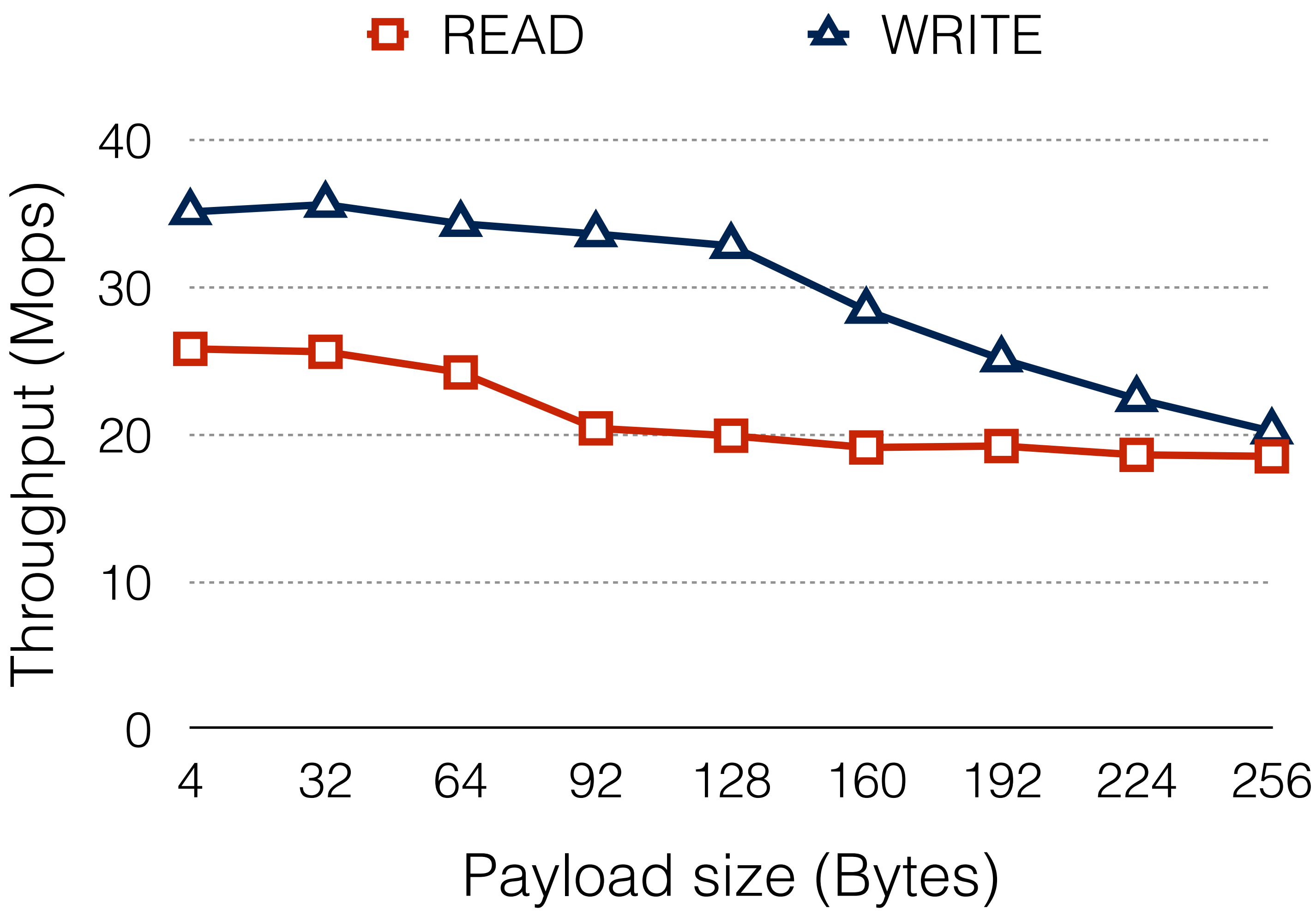
✓ Lower latency ? High throughput

RDMA WRITEs faster than READs



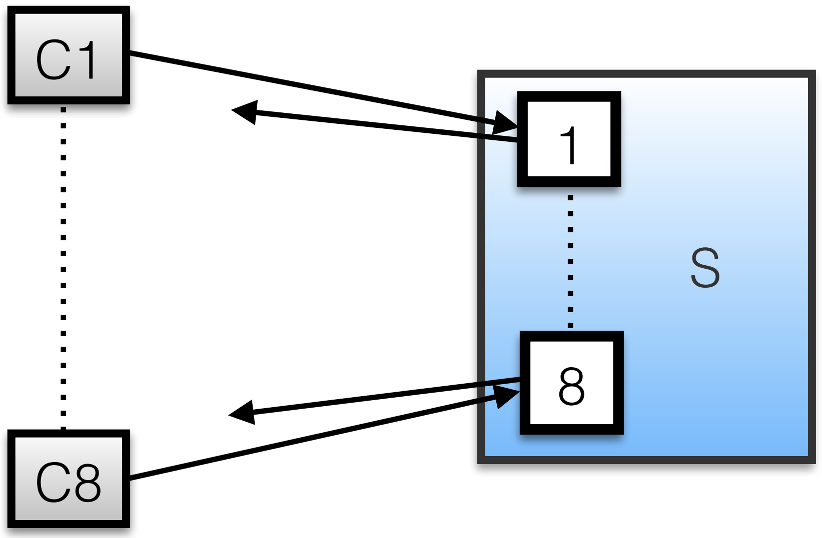
Setup: Apt Cluster

192 nodes, 56 Gbps IB

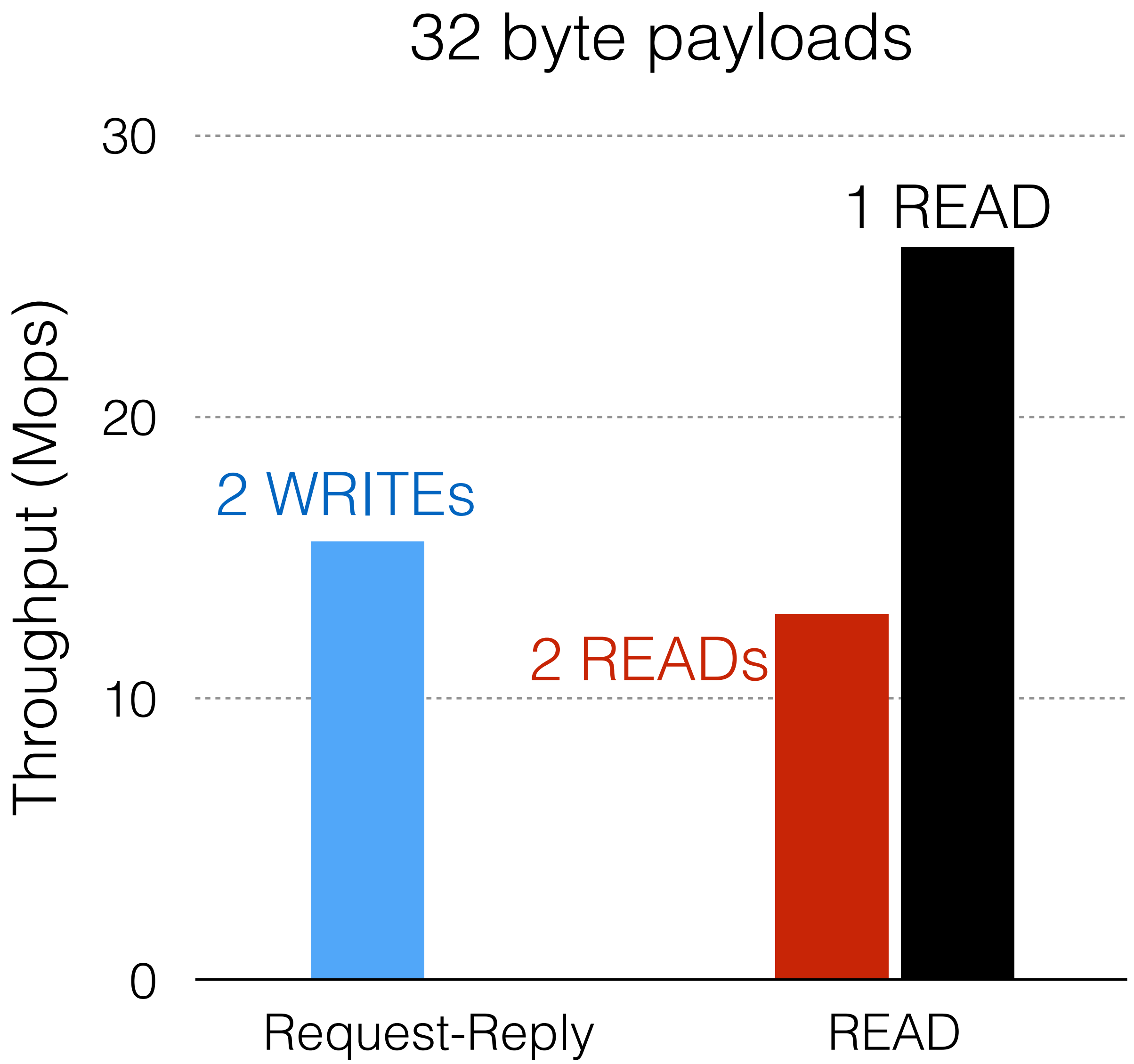


High-speed request-reply

Request-reply throughput:



Setup: one-to-one client-server communication



Step 2:
Optimize the primitives
(details in paper)

Key takeaway: *Naive* uses of other RDMA primitives are slow
But there exist *optimized* uses that are really fast

Evaluation

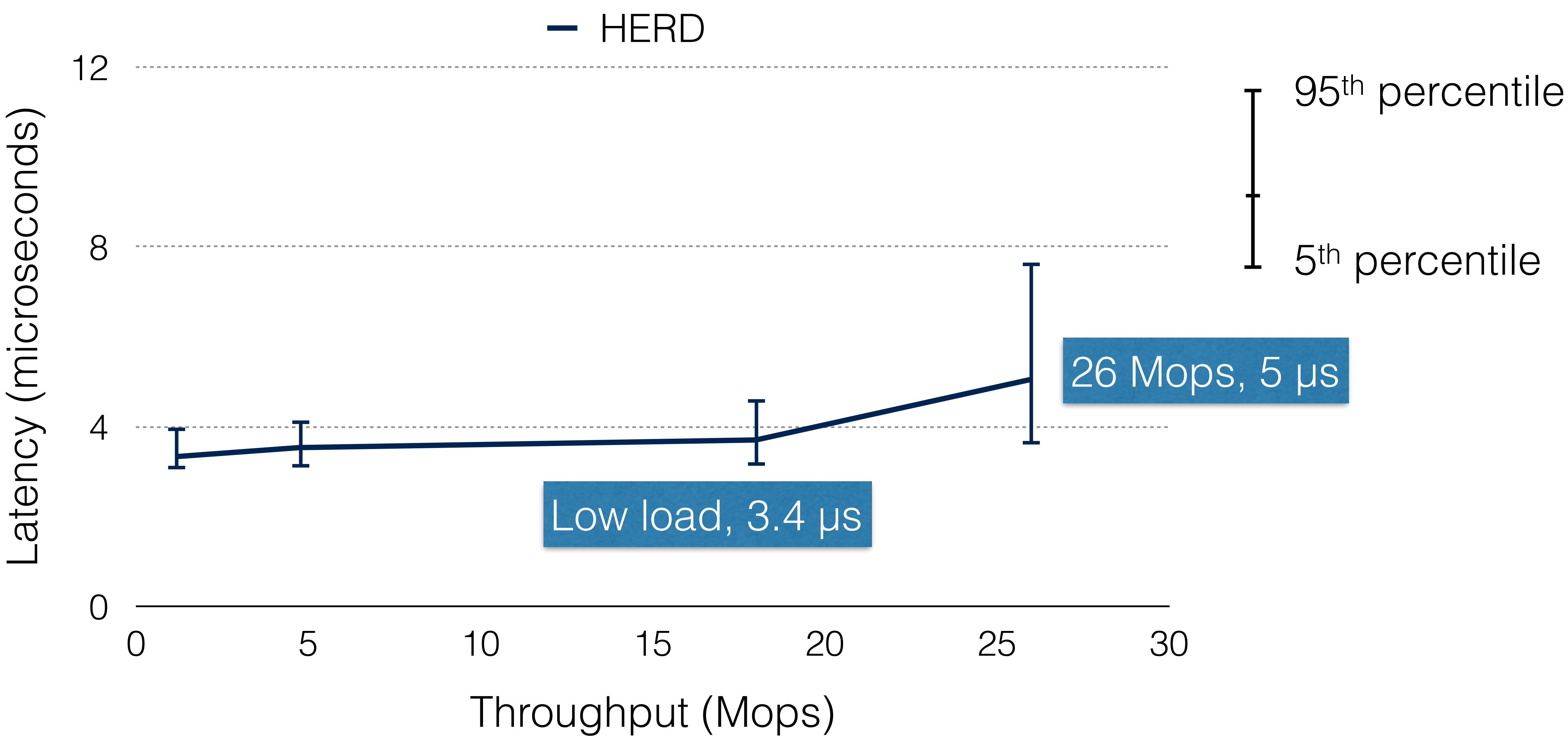
HERD = Request-Reply + MICA [NSDI 2014]

Compare against emulated versions of Pilaf and FaRM-KV

- No datastore
- Focus on maximum performance achievable

Latency vs throughput

48 byte items, GET intensive workload

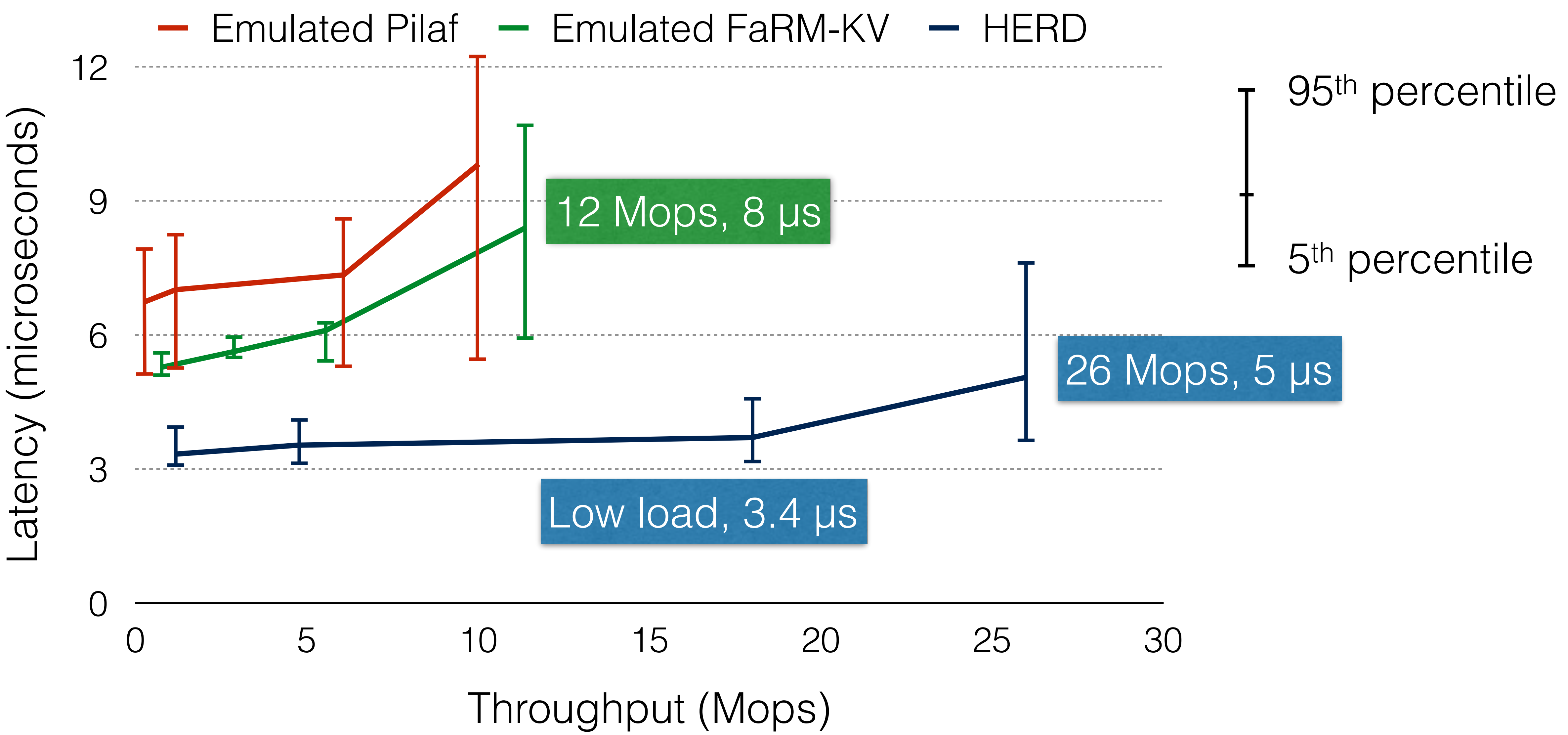


Low load, 3.4 μs

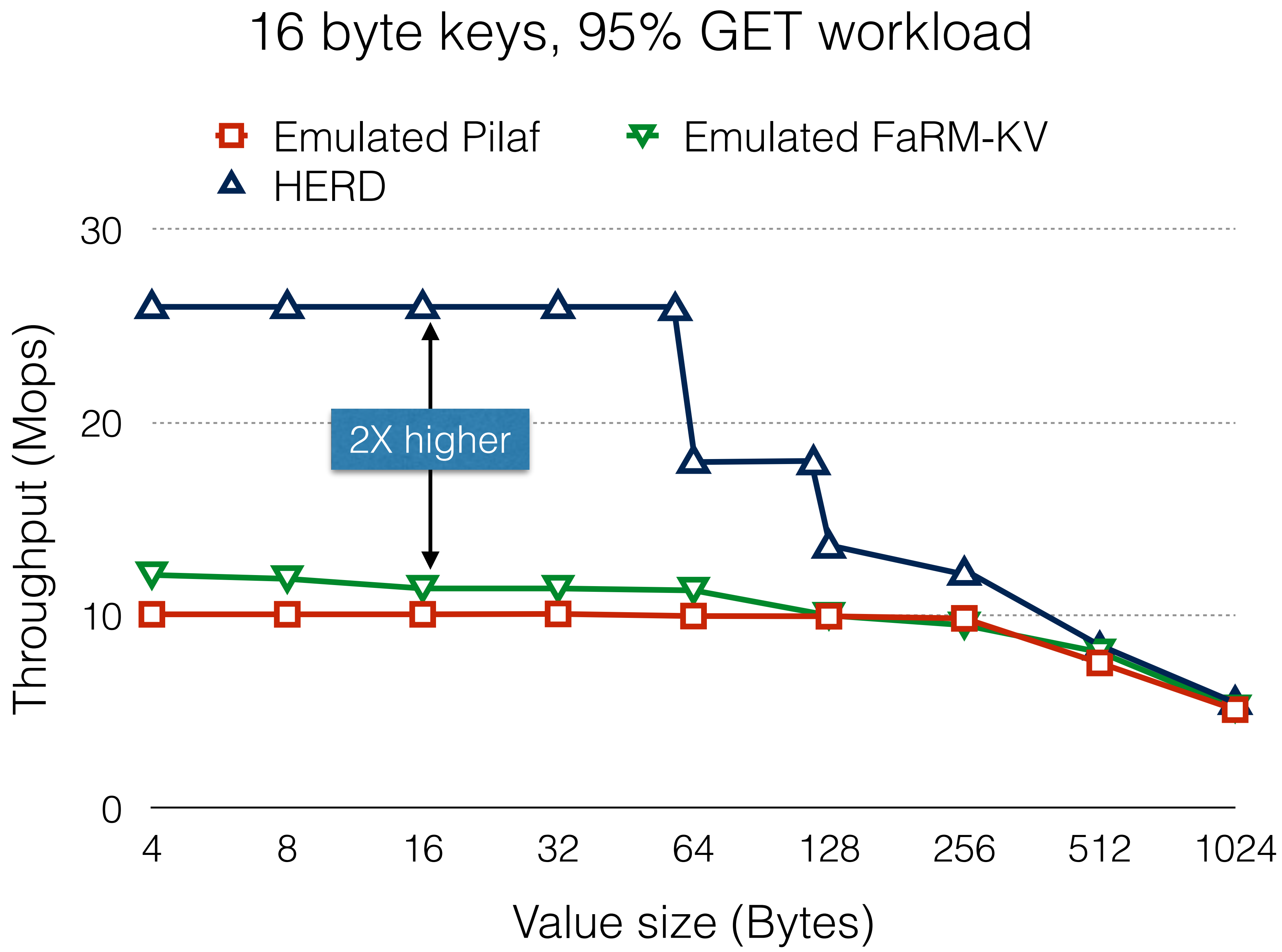
26 Mops, 5 μs

Latency vs throughput

48 byte items, GET intensive workload



Throughput comparison



Computational Efficiency

Memory Efficiency

MICA and HERD key-value stores

This is hard.
Can we (semi) automate?

Algorithmic Optimization

Architectural Tailoring

Good Data Structures

Protocols that are locality-friendly

Optimize for the right things (few RTTs!)