USENIX
ATC '22

AINiCo: SmartNIC-accelerated Contention-aware Request Scheduling for Transaction Processing

Junru Li, Youyou Lu, Qing Wang, Jiazhen Lin, Zhe Yang, Jiwu Shu

Tsinghua University

Transaction Request Scheduling

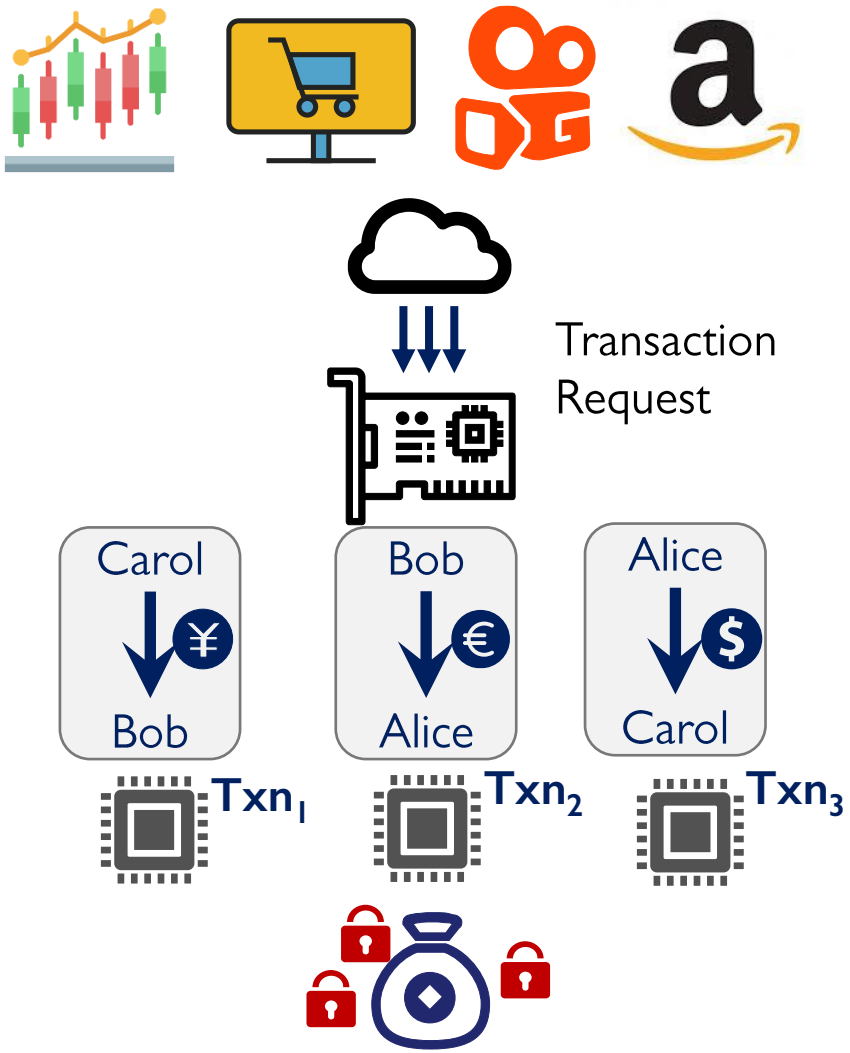
Conflicting transactions are common

- ❖ Stock Exchange: popular stocks
- ❖ Live Selling: popular products

When they concurrently run in different threads

Contention

- ❖ Costly transaction aborts/blocking
  - ❖ Degrade performance and waste CPU resources
  - ❖ More serious with modern multicore servers



Solution: schedule conflicting transactions to the same threads

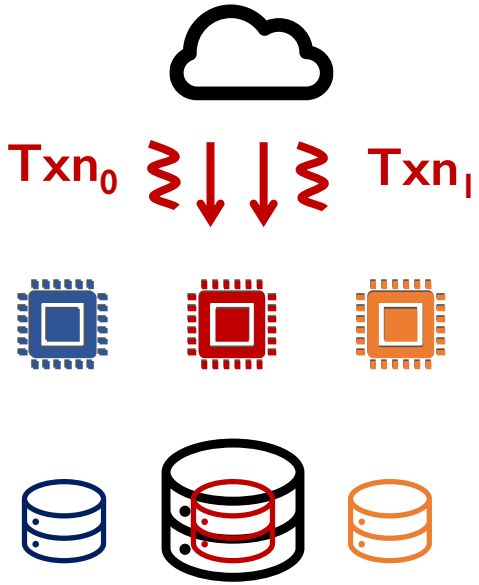
Existing Scheduling Methods

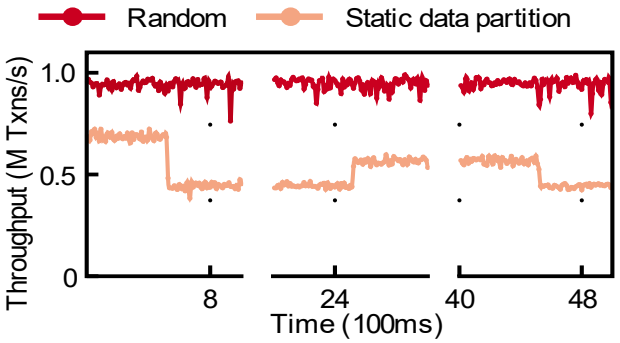
Static data partitioning

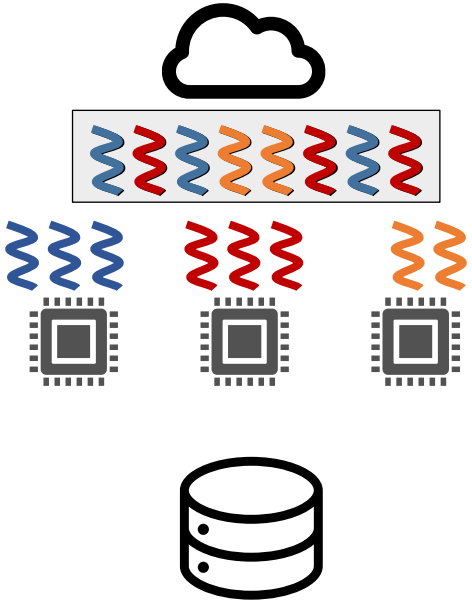
- ❖ Each thread manages a data partition
- ✓ Low latency
- × Not support dynamic workloads

Batching-based scheduling

- ❖ Batching & Grouping
- ✓ Support dynamic workloads
- × High latency for batching



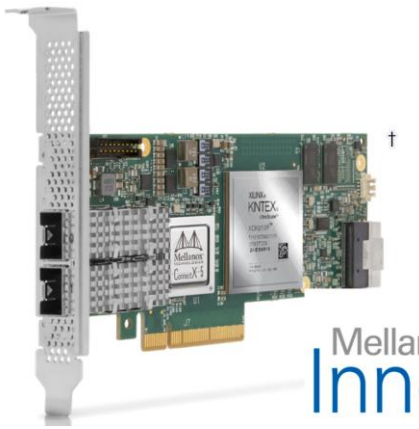




≥ 5ms in Strife [SIGMOD'20]

How to make the scheduler support **dynamic workloads** while **keeping latency low**?

Opportunities from FPGA-based SmartNICs



Mellanox Innova 2 FLEX

NVIDIA

Mellanox TECHNOLOGIES

XILINX

FPGA-based SmartNIC

- Tracking all in/out packets
- Equipped with a full-fledged network ASIC
- Equipped with an FPGA
- …

Using FPGA-based SmartNICs to design a transaction scheduler is promising …

To schedule a transaction

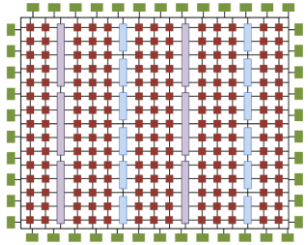
- Multiple keys in the transaction
- Multiple candidate threads

Have opportunities to leverage data parallelism in FPGAs

Challenges

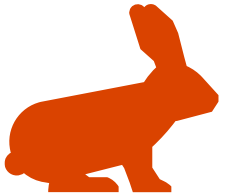
Restricted expressive power

How to map transaction scheduling logic into FPGA ?



Dynamic workloads

How to react to workload changes quickly?



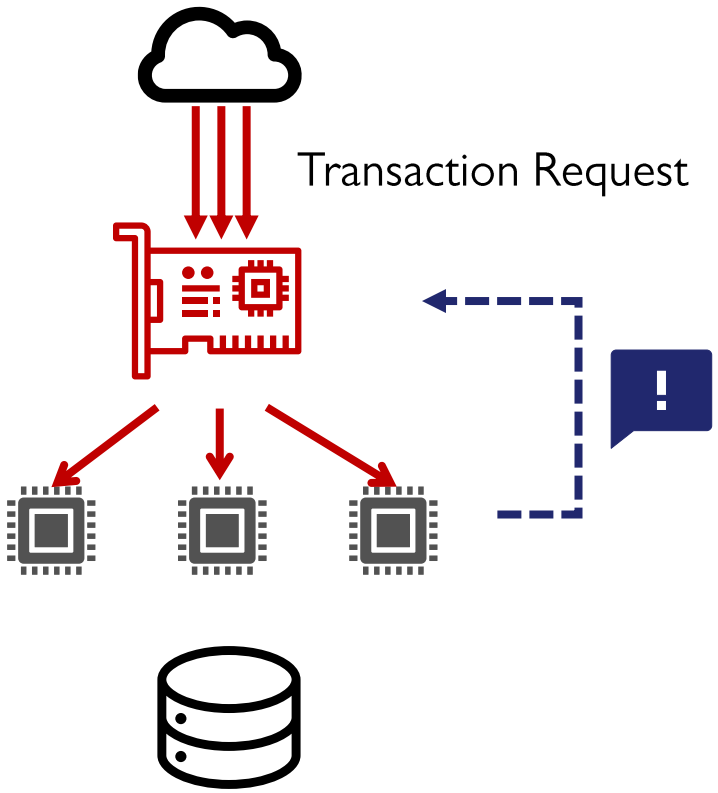
Outline

- Background & Motivation
- AlNiCo: a Contention-aware Transaction Scheduler
- Results
- Summary

Overview

On-SmartNIC Contention-aware Transaction Scheduler

- On-SmartNIC scheduling algorithm
- Software feedback mechanism



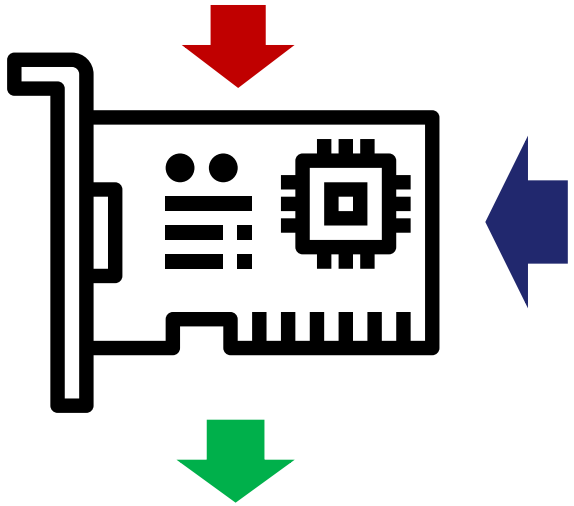
Overview

Contention-aware Transaction Scheduler

Request: where should I go?

Request state

to describe the resources required by a transaction



Worker state

to describe the resources that workers are accessing or will access

Global state

to describe workload characteristics (e.g., hotspots)

Scheduler: you should be sent to worker-i.

Key Designs

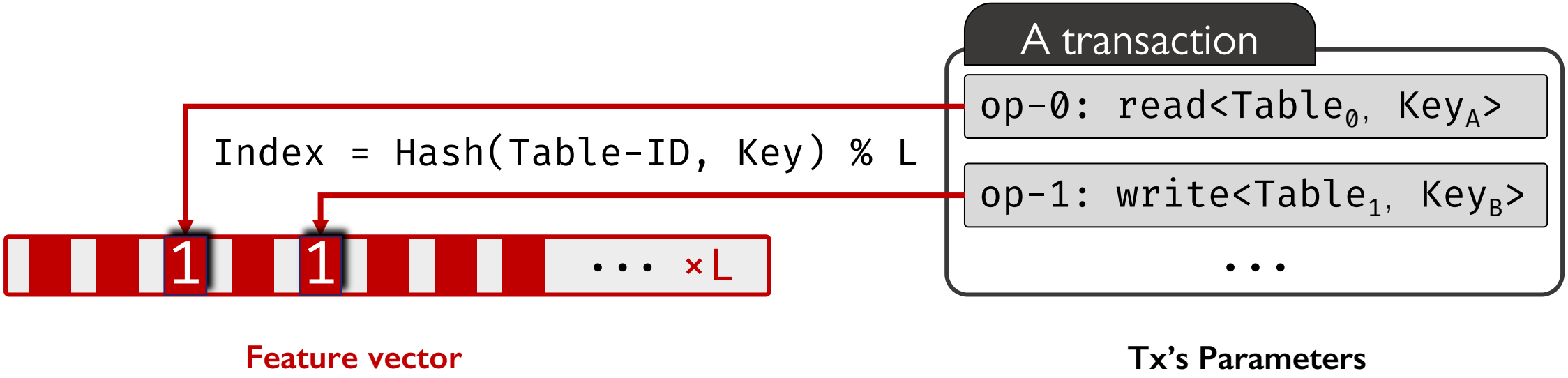
- Restricted expressive power
- States for scheduling → Vectors
- Scheduling algorithm → Vector computation

- Dynamic workloads
- Software feedback mechanism

States for Scheduling: Vectors (1)

States for scheduling: request state, worker state, global state

- **Request state**
  - Request feature vector

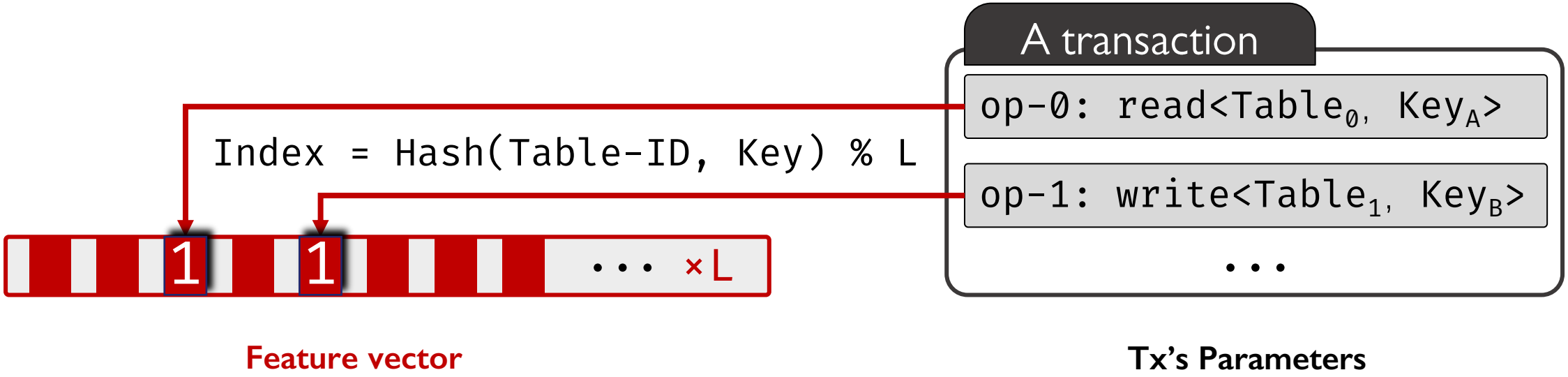


Request Format

States for Scheduling: Vectors (1)

States for scheduling: request state, worker state, global state

- **Request state**
  - Request feature vector



New Header

States for Scheduling: Vectors (1)

States for scheduling: request state, worker state, global state

- **Request state**
  - Request feature vector

```
Index = Hash(Table-ID, Key) % L
```

Requirements

- **Goal: to avoid hash collisions**
- Keys of different tables should be mapped into different features
- The number of features of a table should be proportional to its size

A transaction

```
op-0: read<Table0, KeyA>
op-1: write<Table1, KeyB>
...
```

Tx's Parameters

Feature vector

States for Scheduling: Vectors (1)

States for scheduling: request state, worker state, global state

- **Request state**
  - Request feature vector

```
Index = Hash(Table-ID, Key) % L
```

Corner cases

- **Range queries:** randomly generate the keys in the ranges
- **Non-primary keys:** maintain a secondary-index cache

A transaction

```
op-0: read<Table0, KeyA>
op-1: write<Table1, KeyB>
...
```

Tx's Parameters

Feature vector

States for Scheduling: Vectors (II)

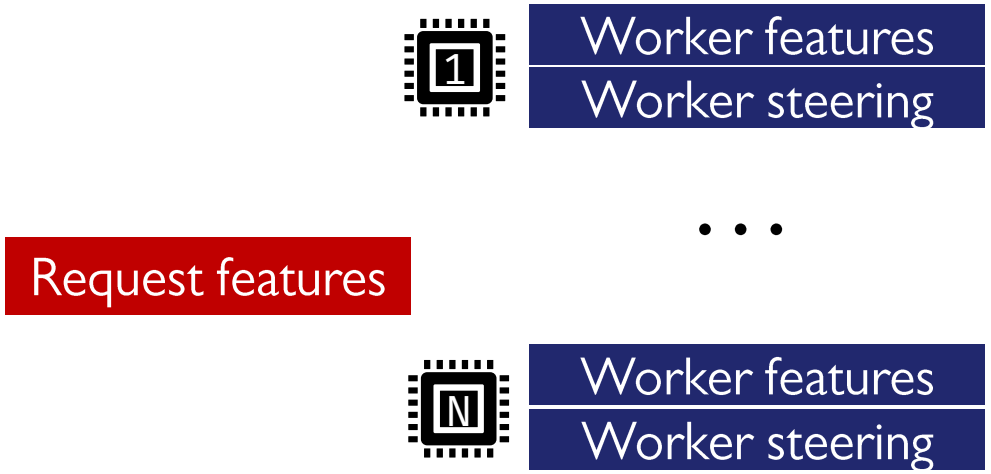
States for scheduling: request state, worker state, global state

- **Request state**
  - **Request feature vector** (generated by clients)
- **Worker state**
  - Each worker has a **worker feature vector**: ongoing Txns' feature vectors(updated by FPGA)
  - Each worker has a **steering vector**: steering rules (updated by software)
- **Global state**
  - A **weight vector**: feature's hotness (update by software)

Scheduling Algorithm: Vector Computation

Making scheduling decisions with FPGA acceleration

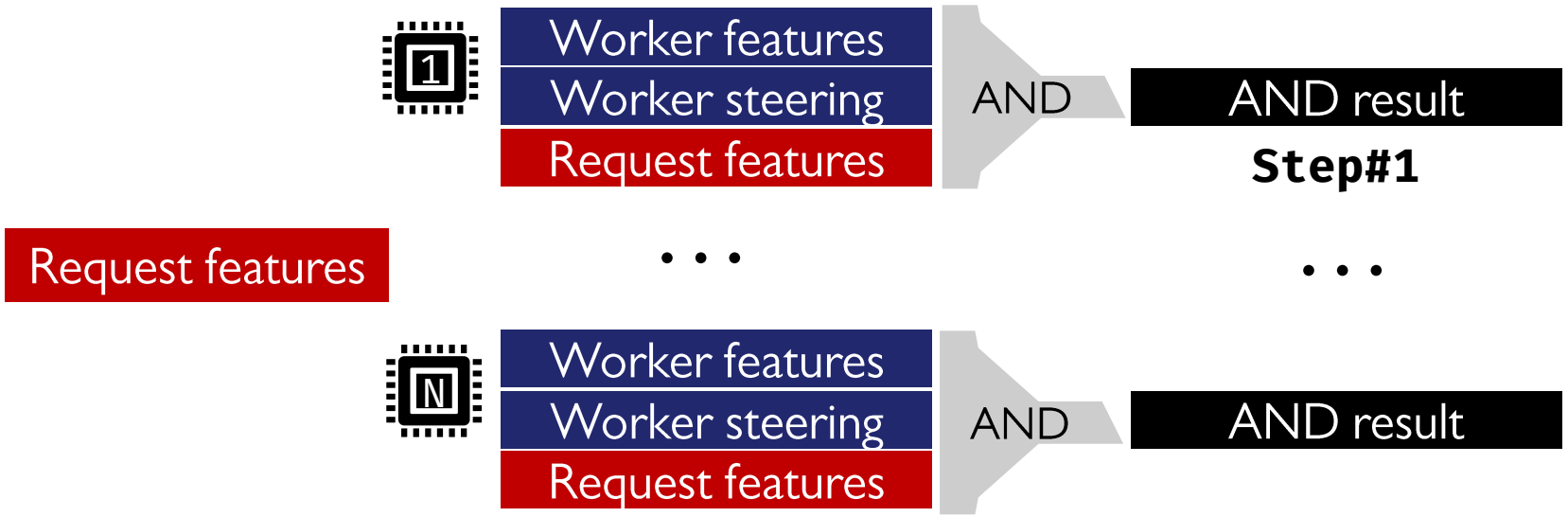
- Step#1: get the same features between the new request and each worker



Scheduling Algorithm: Vector Computation

Making scheduling decisions with FPGA acceleration

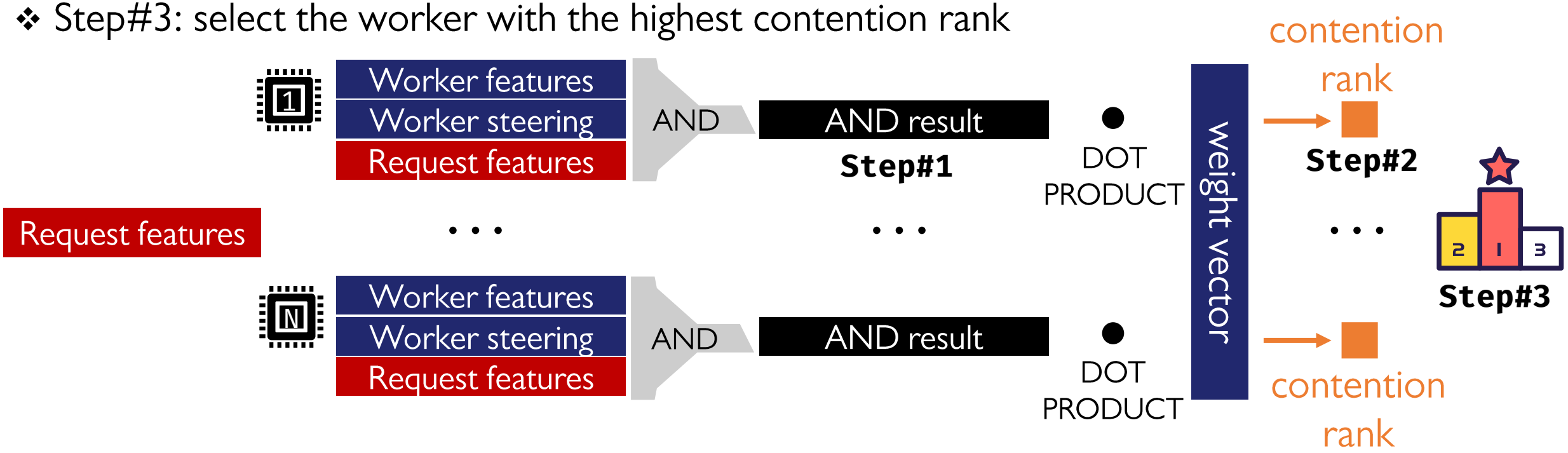
- Step#1: get the same features between the new request and each worker



Scheduling Algorithm: Vector Computation

Making scheduling decisions with FPGA acceleration

- Step#1: get the same features between the new request and each worker
- Step#2: calculate a **contention rank** for each worker
- Step#3: select the worker with the highest contention rank



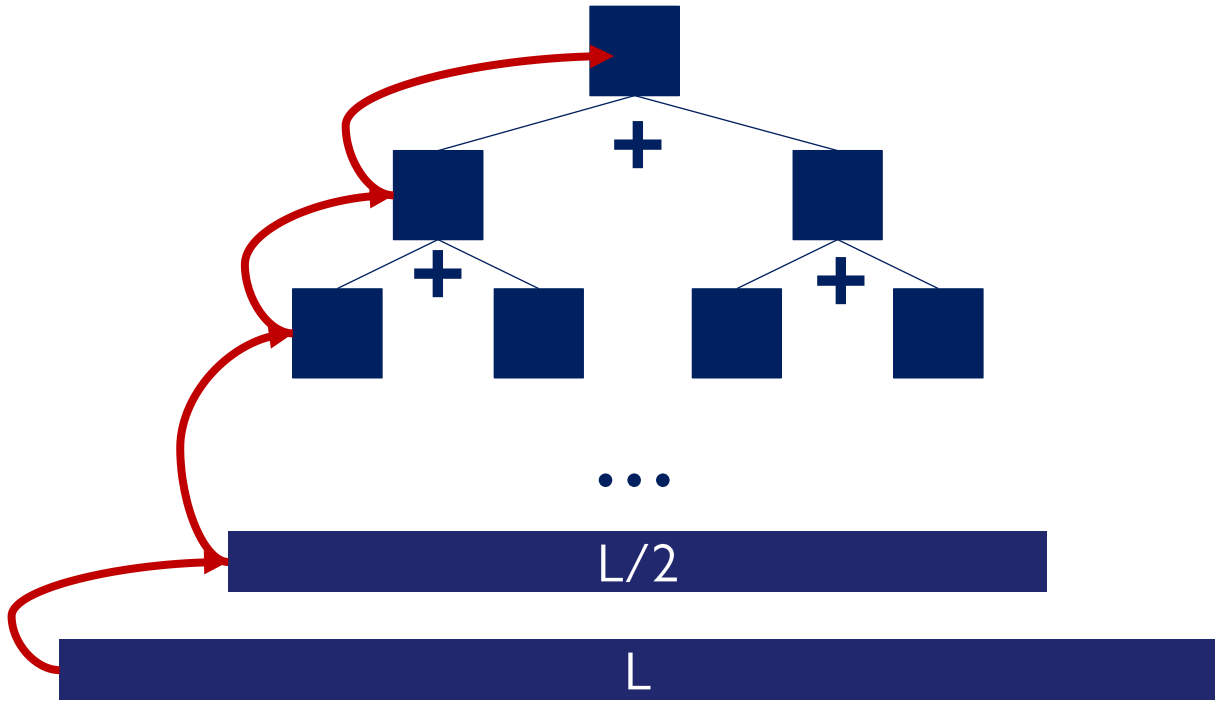
Scheduling Algorithm: Vector Computation

Making scheduling decisions with FPGA acceleration

- Step#1: get the same features between the new request and each worker
- Step#2: calculate a **contention rank** for each worker
- Step#3: select the worker with the highest contention rank

FPGA acceleration

- Step#1: Data parallelism
- Step#2: Sum reduction
- Step#3: Max reduction



Sum reduction

Software Feedback Mechanism

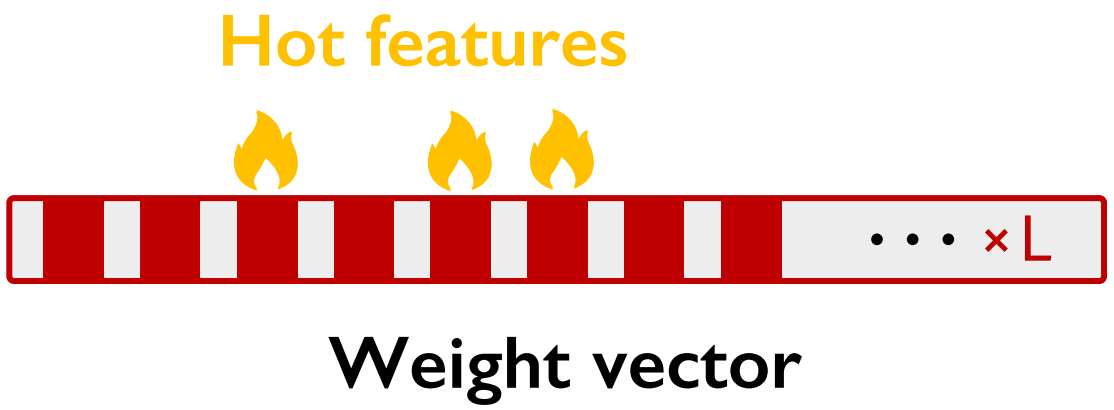
Worker threads periodically update the states in the hardware

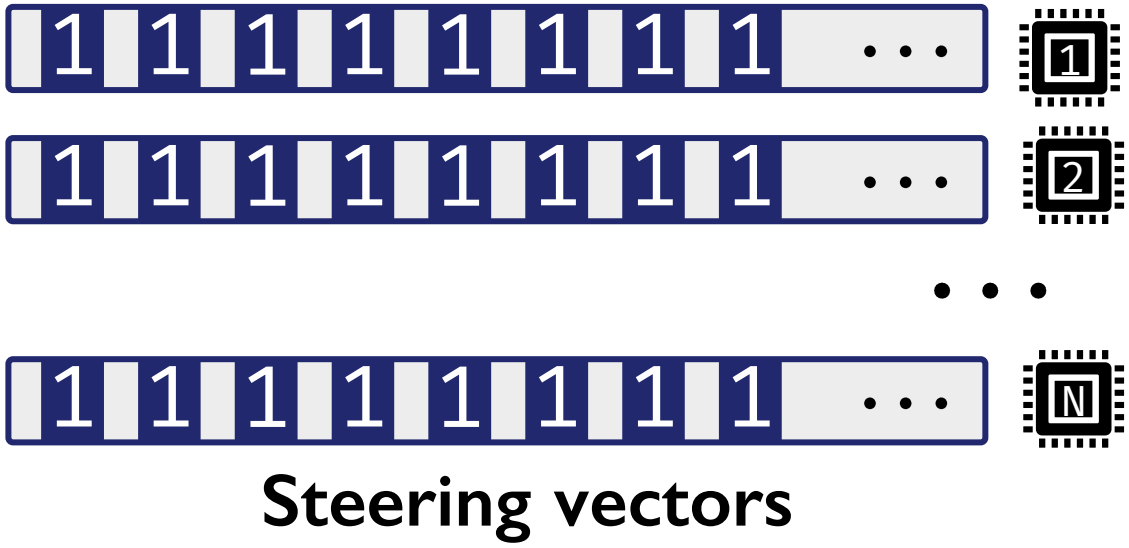
- **Weight vector**
  - Goal: to describe the hotspots in transaction systems
  - The keys that **cause contention frequently** are hotspots and have higher weights
- **Steering vector**
  - Goal: to schedule transactions with less contention to different workers

Weight vector

Software Feedback Mechanism

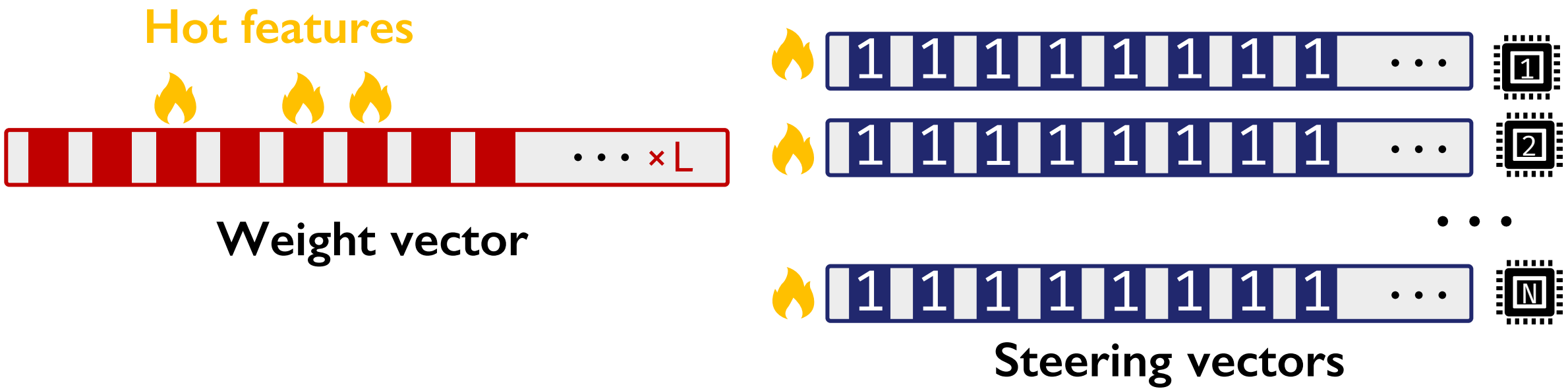
- **Steering vector**
  - Goal: to schedule transactions with less contention to different workers
  - Select the hottest N features





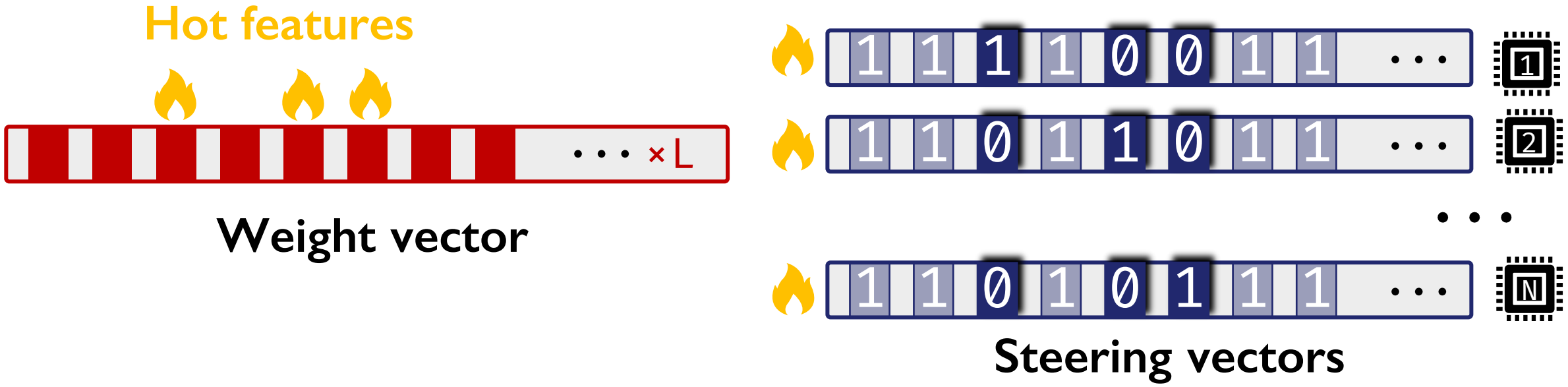
Software Feedback Mechanism

- **Steering vector**
  - Goal: to schedule transactions with less contention to different workers
  - Select the hottest N features
  - Assign these N features to different workers



Software Feedback Mechanism

- **Steering vector**
  - Goal: to schedule transactions with less contention to different workers
  - Select the hottest N features
  - Assign these N features to different workers
  - Each worker steers a hot feature exclusively

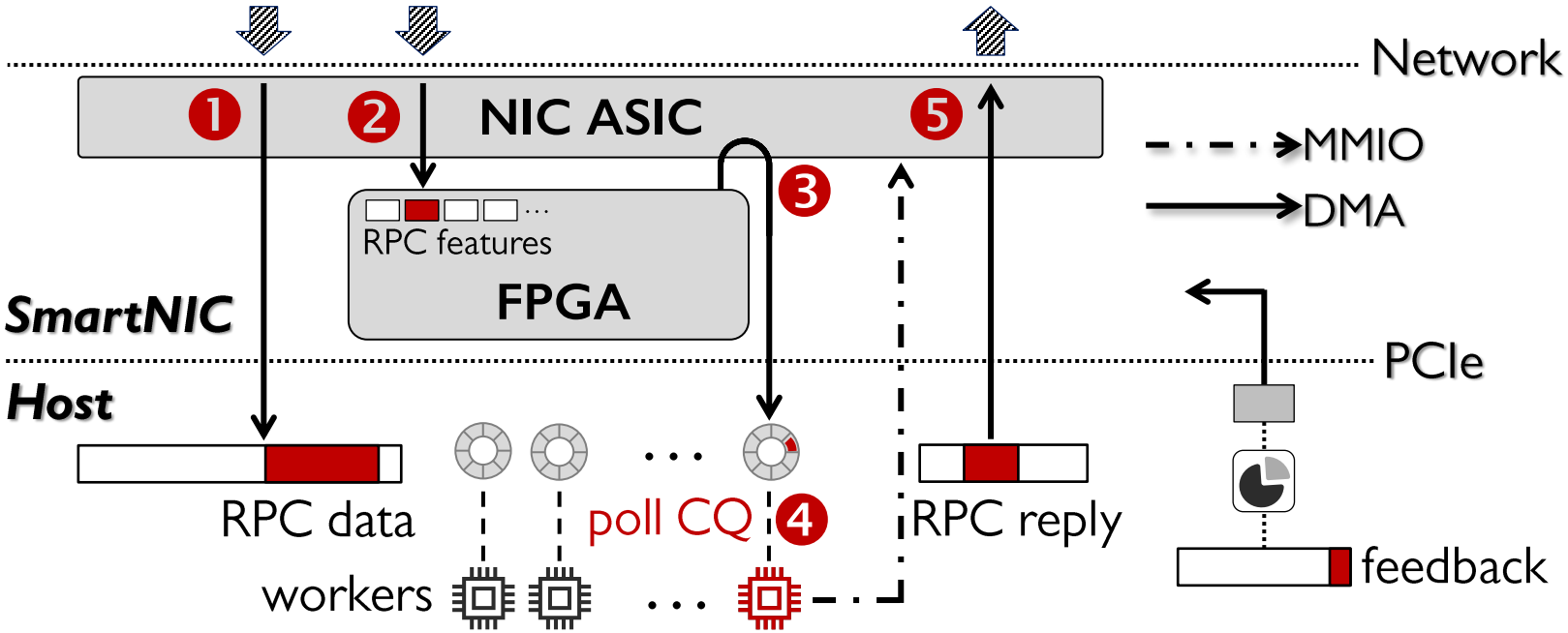


- **The FPGA fetches weight vector and steering vectors via DMA**

More Details: checkout our paper

- **Other design details**
  - How to describe the read/write modes in features
  - When to update the weight vector and steering vectors
  - How to reserve CPU cores for long running transactions via AlNiCo
  - How to support various CC protocols
  - …

Implementation on Mellanox Innova-2



A scheduling-enabled RDMA RPC

❶ The client writes **RPC data** to the host (RDMA write)

❷ The client writes **the feature vector** to the FPGA (RDMA write)

❸ The FPGA schedules the request and writes a worker's CQ (DMA)

❹ The Worker polls a CQ entry and executes the corresponding transaction

❺ The Worker sends a reply to the client (RDMA write)

Fundamental drivers

- NIC to FPGA (PCIe P2P)
- FPGA to HOST (DMA)

Outline

- Background & Motivation
- AlNiCo: a Contention-aware Transaction Scheduler
- Results
- Summary

Experimental Setup

Hardware Platform

CPU	2 Intel 12-core Xeon E5-2650 CPUs
Client's NIC	100Gbps Mellanox ConnectX-5
Server's NIC	25Gbps ×2 Mellanox Innova-2

Configuration

FPGA frequency	250MHz
Feedback epoch	20ms
Vector length	512

Competitors

NetSTO	RDMA-based RPC + **STOv2** *[VLDB'20]*
StaticPart	Static data partitioning, TPC-C: warehouse ID, YCSB: table ID
Strife *[SIGMOD'20]*	Batching-based scheduling, batch size: 10K transactions or 5ms

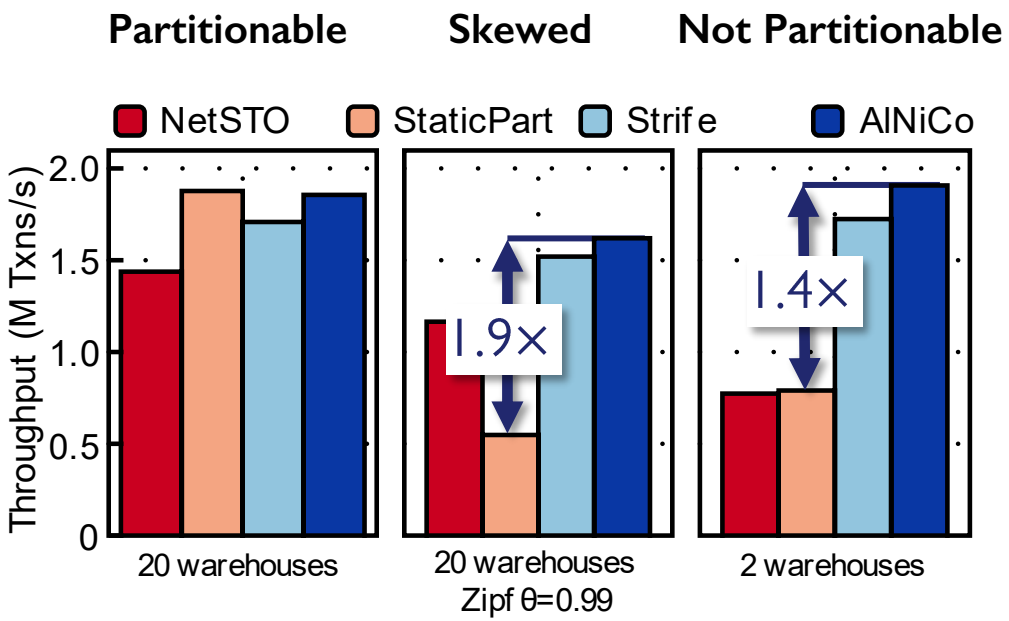
Benchmarks: TPC-C, YCSB-T, TCSB-HOT

CC Protocols: Silo (default configuration), TicToc, Cicada, 2PL

Overall Performance (Throughput)

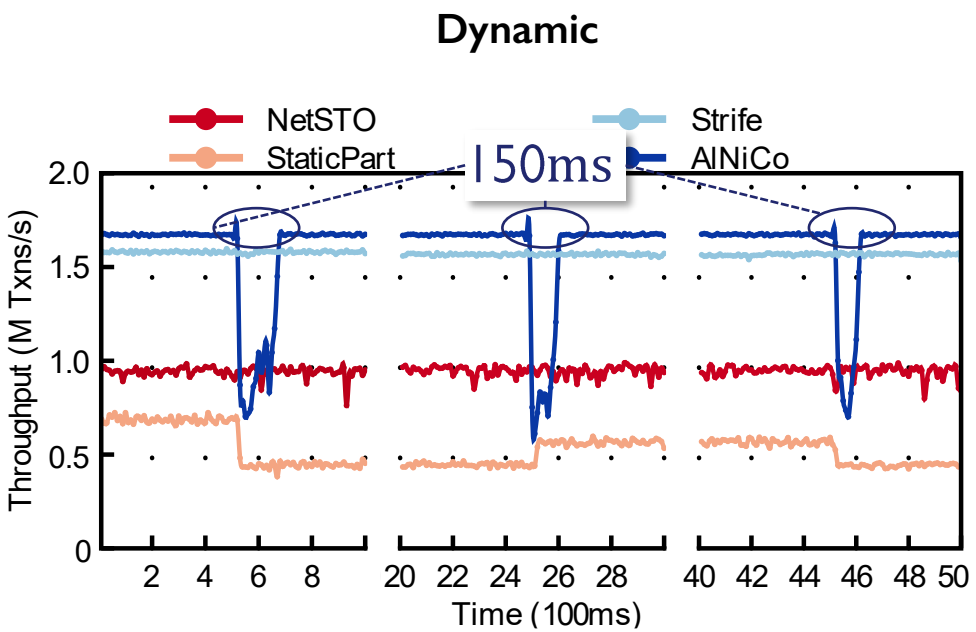
TPC-C

- 20 workers for normal transactions
- 2 workers for long running transactions



YCSB-HOT

- 16 keys in each transaction
- write-read ratio: 50/50
- hotspots change every 2s

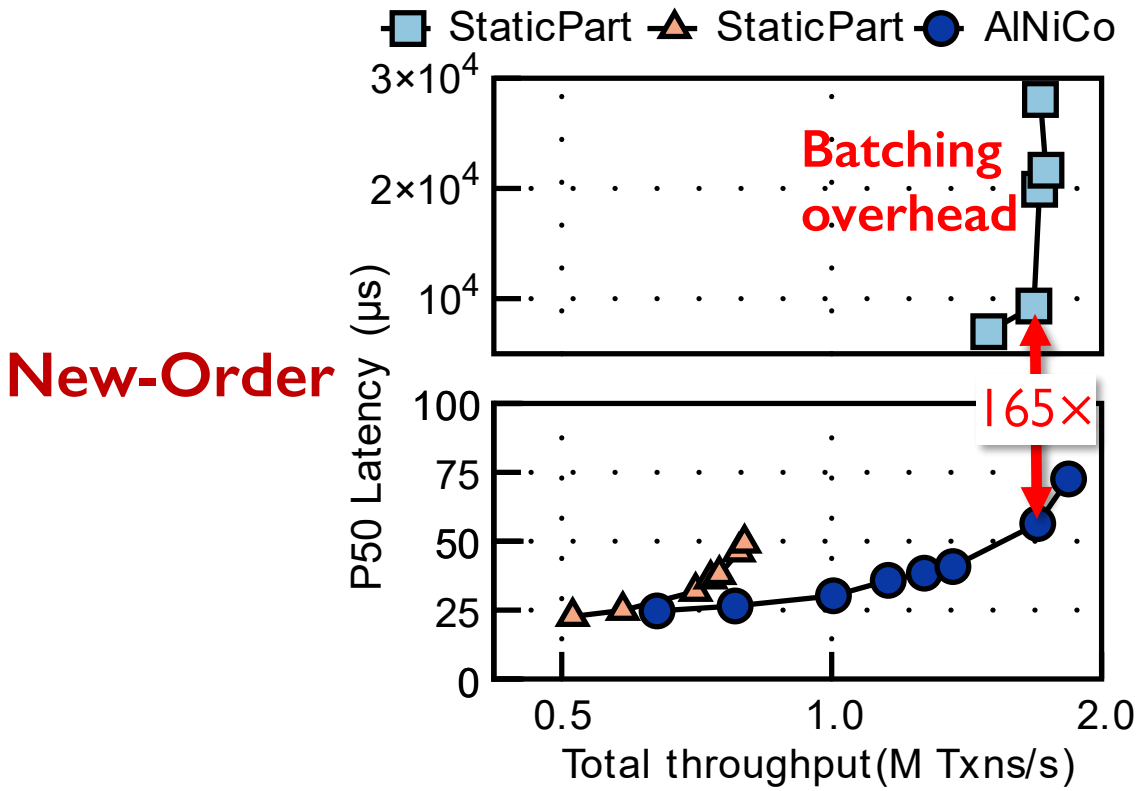


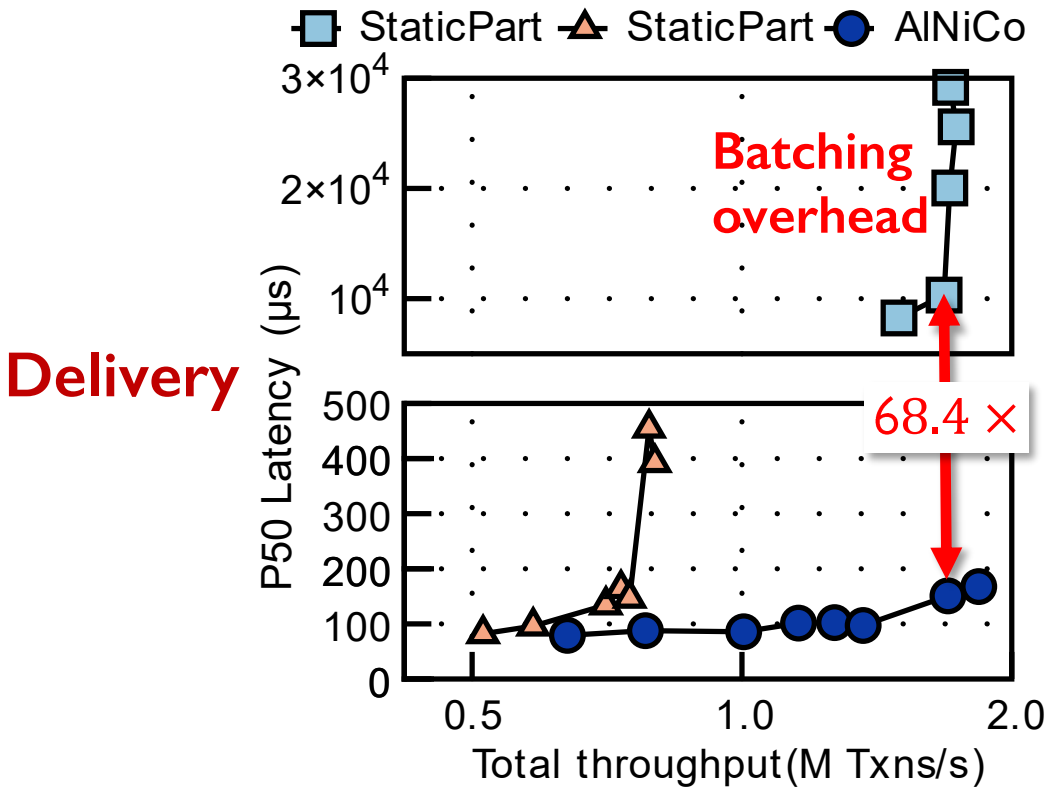
AlNiCo effectively improves throughput and reacts quickly to hotspot changes.

Overall Performance (Latency)

TPC-C Latency

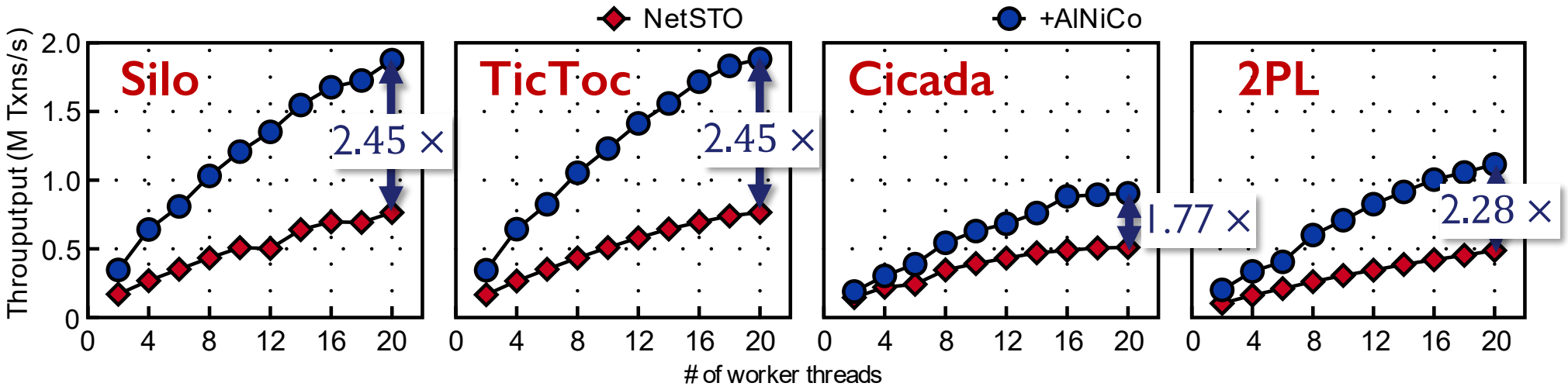
- 2 warehouses (high-contention, not partitionable)





AlNiCo does not introduce extra latency for scheduling.

Generality



Generality of AINiCo

TPC-C, 2 warehouses

- CC protocols: Silo, TicToc, Cicada, 2PL


Speedy Transactions in Multicore In-Memory Databases

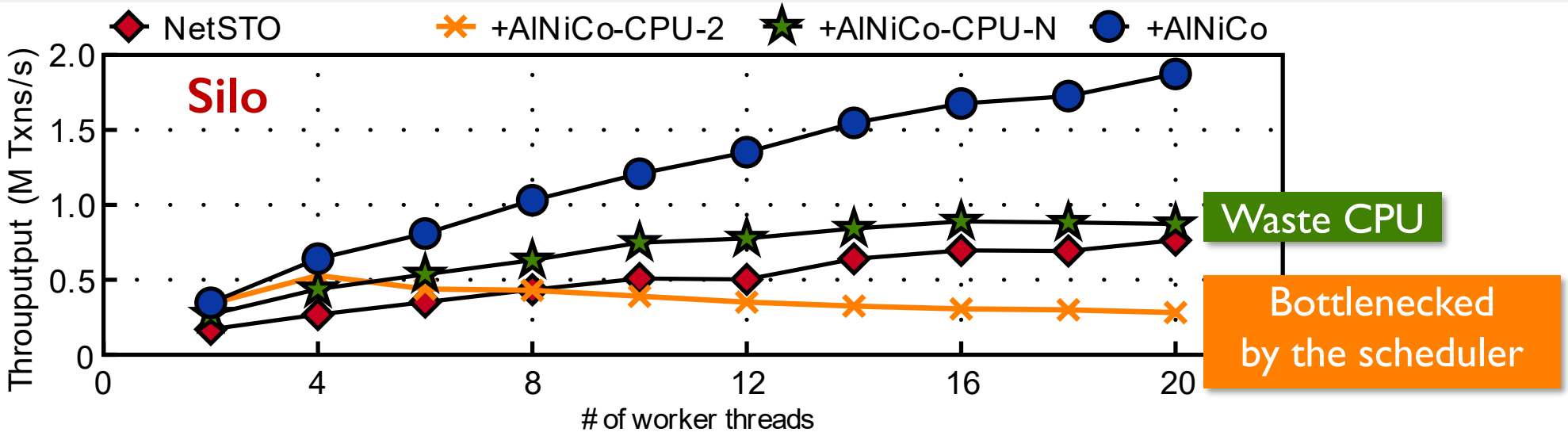
TicToc: Time Traveling Optimistic Concurrency Control

Cicada: Dependably Fast Multi-Core In-Memory Transactions

Hyeontaek Lim, Carnegie Mellon University, hl@cs.cmu.edu
Michael Kaminsky, Intel Labs, michael.e.kaminsky@intel.com
David G. Andersen, Carnegie Mellon University, dga@cs.cmu.edu

AINiCo is generalized for different CC protocols.

Comparison with CPU-version AlNiCo



TPC-C, 2 warehouses, Silo

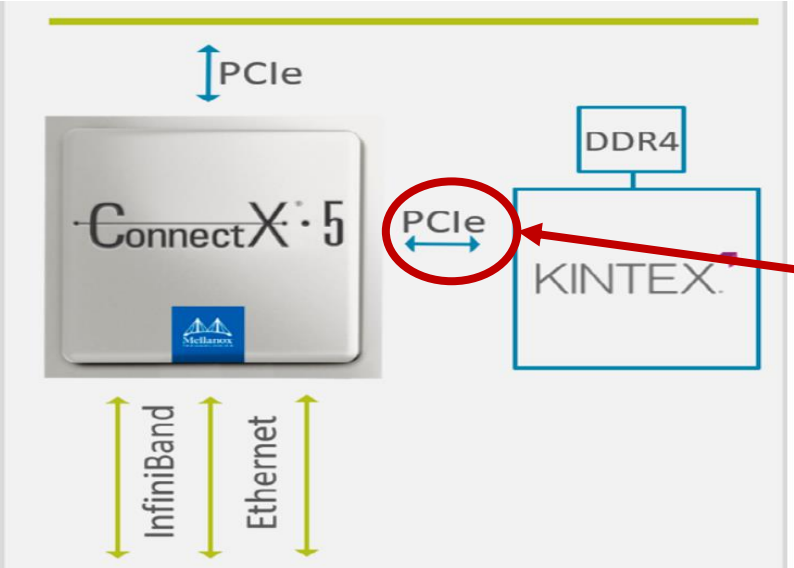
The necessity of SmartNICs

- **AlNiCo-CPU-2**: reserve **two threads** to execute the scheduling logic
- **AlNiCo-CPU-N**: **co-locate** the worker logic and scheduler logic in each thread

The SmartNIC can efficiently reduce scheduling overhead.

Overhead of AlNiCo

The extra latency for PCIe



2 extra
PCIe communications

Median Latency of Null RPC (no contention)

System \ Payload	128	512	2K	4K
NetSTO	3.7µs	4µs	6.7µs	13.4µs
AlNiCo	10.7µs	10.7µs	10.9µs	13.9µs
Delta	**+7µs**	**+6.7µs**	**+3.2µs**	**+0.5µs**

Median latency of TPC-C

New-order
51.8µs
26.5µs
-25.3µs

Outline

- Background & Motivation
- AlNiCo: a Contention-aware Transaction Scheduler
- Results
- Summary

Summary

- **Goal**
  - Scheduling transaction requests to reduce contention with low scheduling latency
- **Key Idea**
  - Using FPGA-based SmartNICs, with hardware-software co-design
- **Techniques in AlNiCo**
  - Describe the transaction scheduling algorithm in a hardware-friendly manner
  - Provide generalized feedback interfaces
- **Results**
  - AlNiCo outperforms state-of-the-arts
  - AlNiCo is generalized for various CC protocols and applications

Thanks

AINiCo: SmartNIC-accelerated Contention-aware Request Scheduling for Transaction Processing

Contact Information: lijr19@mails.tsinghua.edu.cn