Developing and Evaluating the **nanoPU** and **nanoSort** using *Chipyard* and *Firesim*

Stephen Ibanez, Theo Jepsen, Alex Mallery, Serhat Arslan, *Muhammad Shahbaz, Changhoon Kim, Gregory Valiant, Nick McKeown

*Stanford University, *Purdue University*

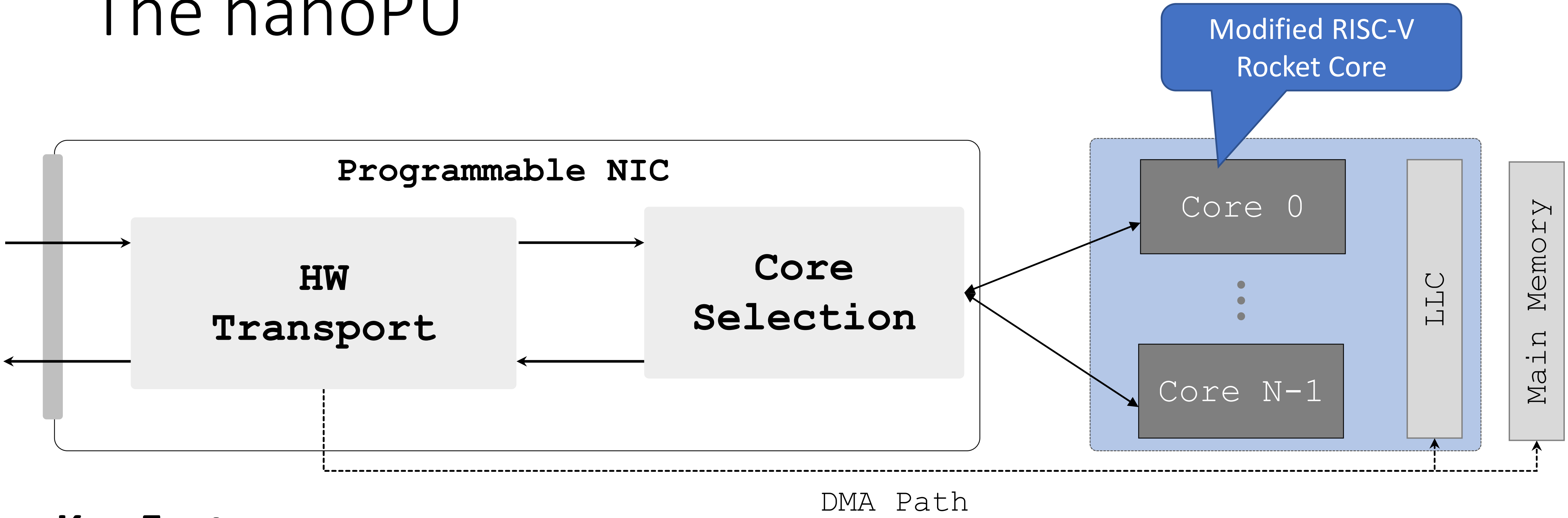
What is the nanoPU? (OSDI ‘21)

Data center applications are *highly distributed* and increasingly *fine grained*.

Question:

What would it take to *absolutely minimize* median and tail communication latency?

The nanoPU

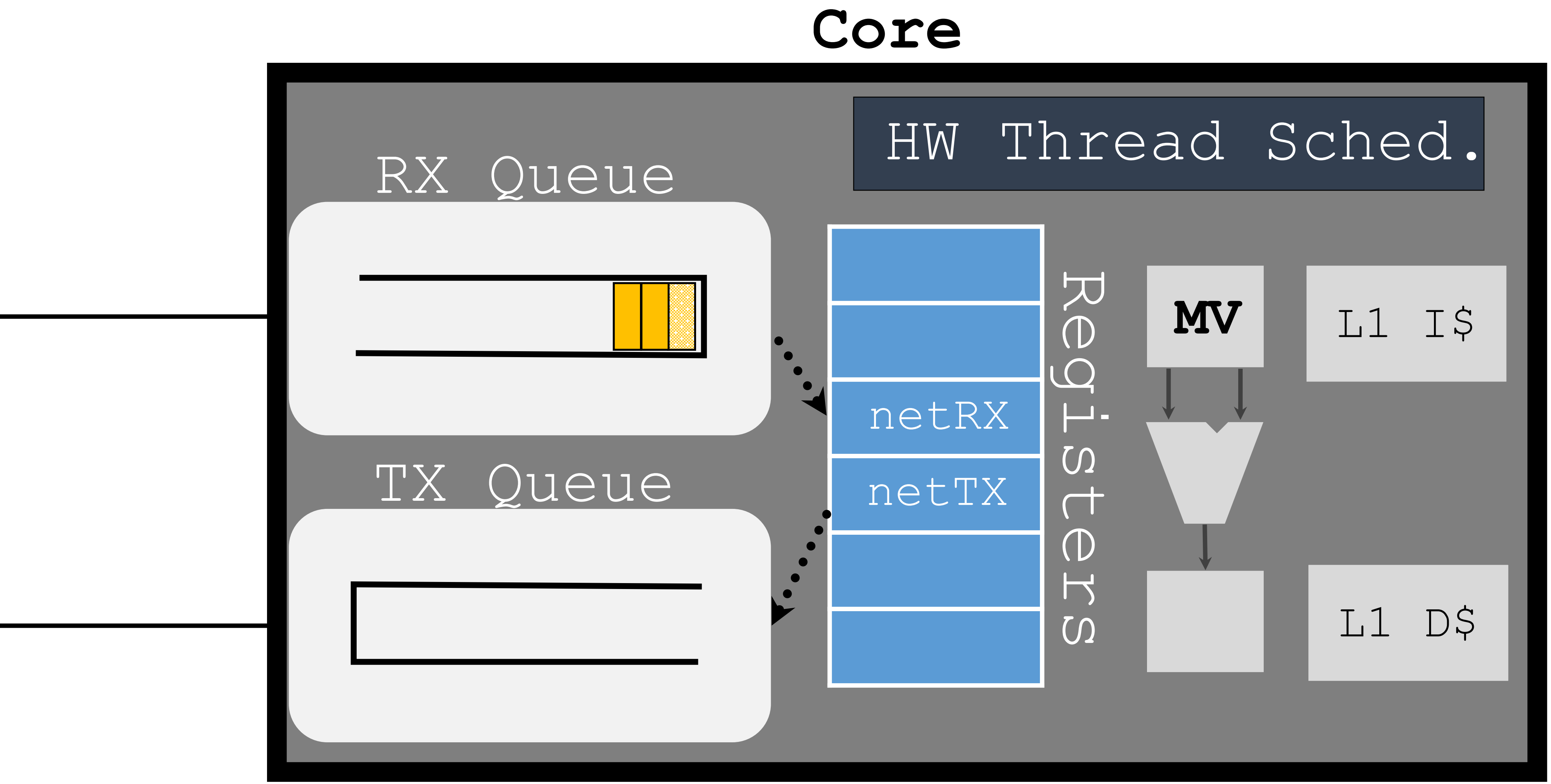


Key Features:

- Integrated NIC
- Efficient core selection in HW
- Programmable transport in HW
- Direct path to CPU register file
- Hardware-accelerated thread scheduling

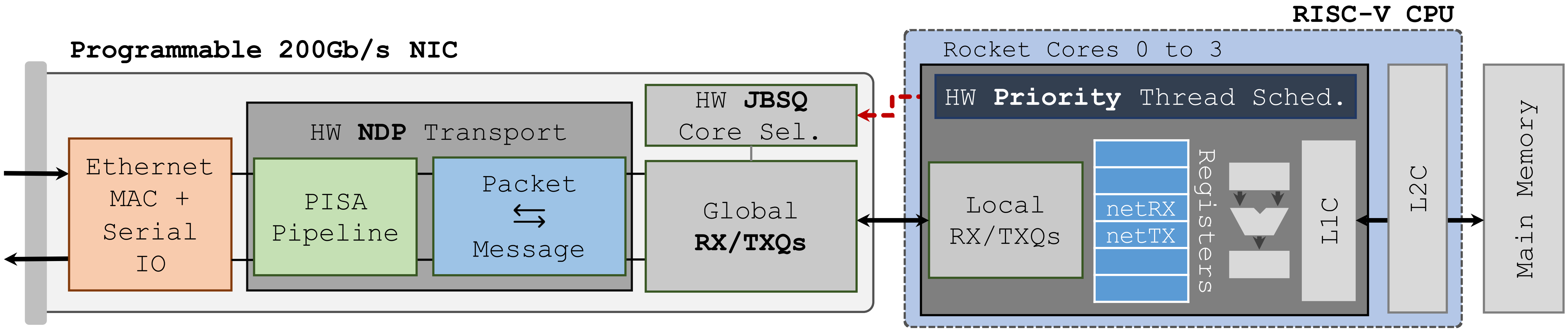
Wire-to-wire latency: **69ns**
Single-core throughput: **118Mrps**

The nanoPU Core

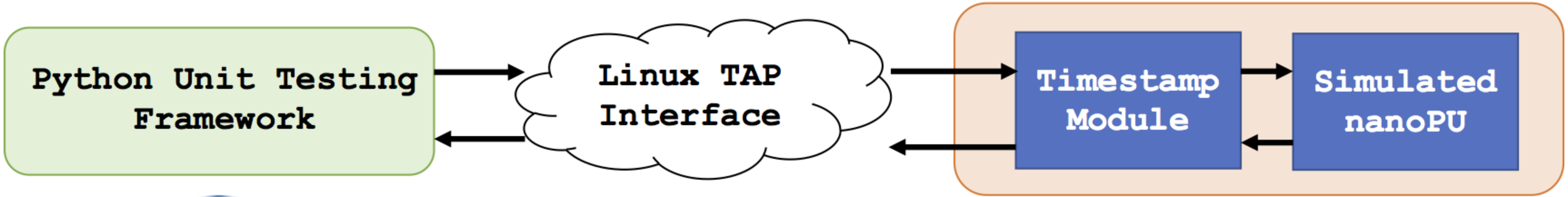


nanoPU Prototype

- Quad-core nanoPU based on RISC-V Rocket core (using Chipyard)
- 4,300 lines of Chisel code & 1,200 lines of C and RISC-V assembly for custom *nano*kernel
- Implements NDP and Homa transport
- Cycle-accurate simulations (3.2GHz) on AWS FPGAs using Firesim



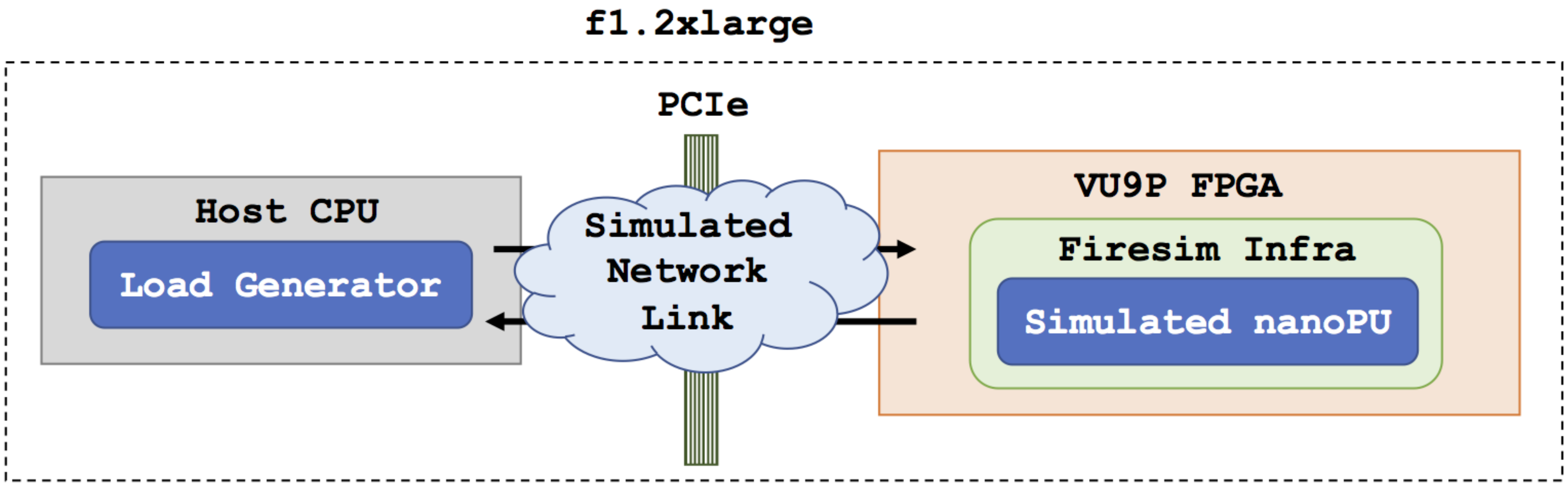
Evaluation Method #1 – Python & Verilator Simulations



VERILATOR

- Hook up Chipyard Verilator simulations to Linux TAP interface
- Leverage powerful Python libraries
  - Scapy – constructing, transmitting, receiving, parsing network pkts
  - Pandas, NumPy – stats collection and numerical analysis
- Used for:
  - Unit testing nanoPU apps
  - Simple latency & throughput microbenchmarks

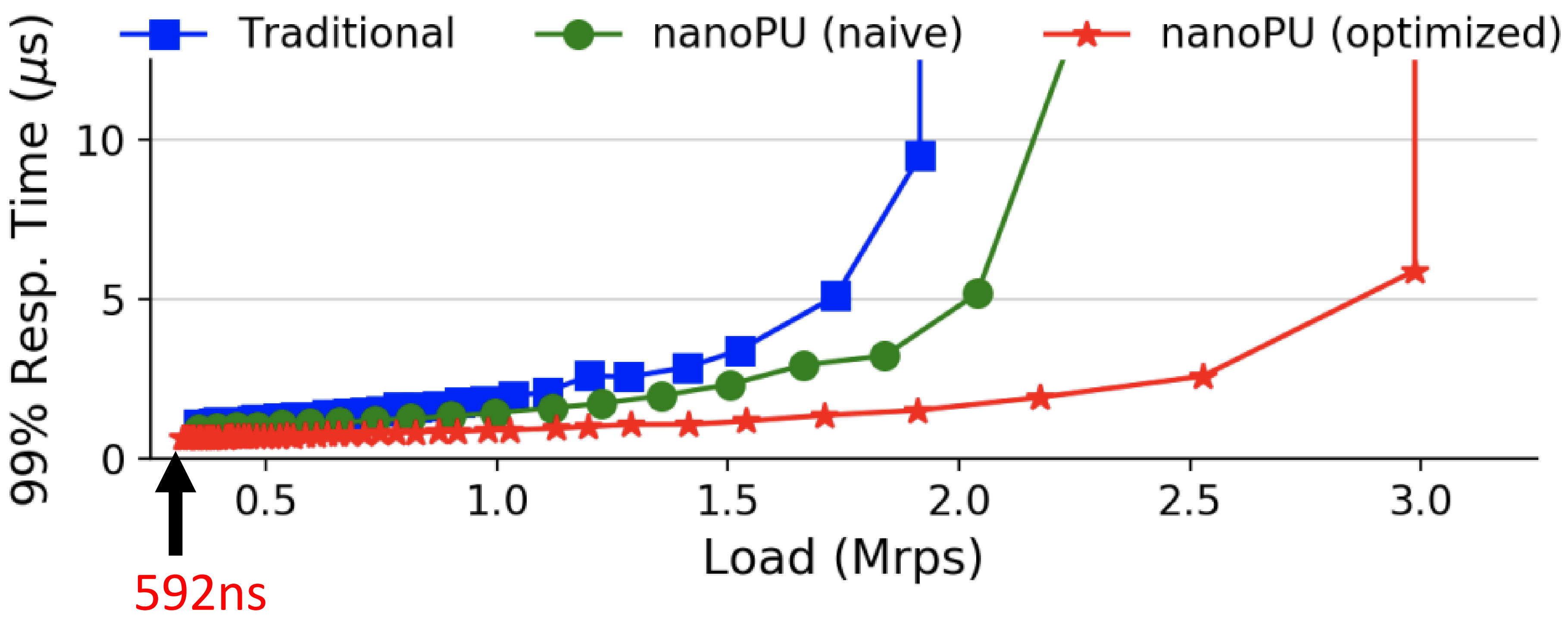
Evaluation Method #2 – Firesim Load Generator



- Firesim load generator
  - A modified Firesim switch model
  - Cycle accurate, running on Host CPU
  - Generates requests at configurable load, measures end-to-end response time
- Firesim enables us to simulate >10K requests in a couple minutes

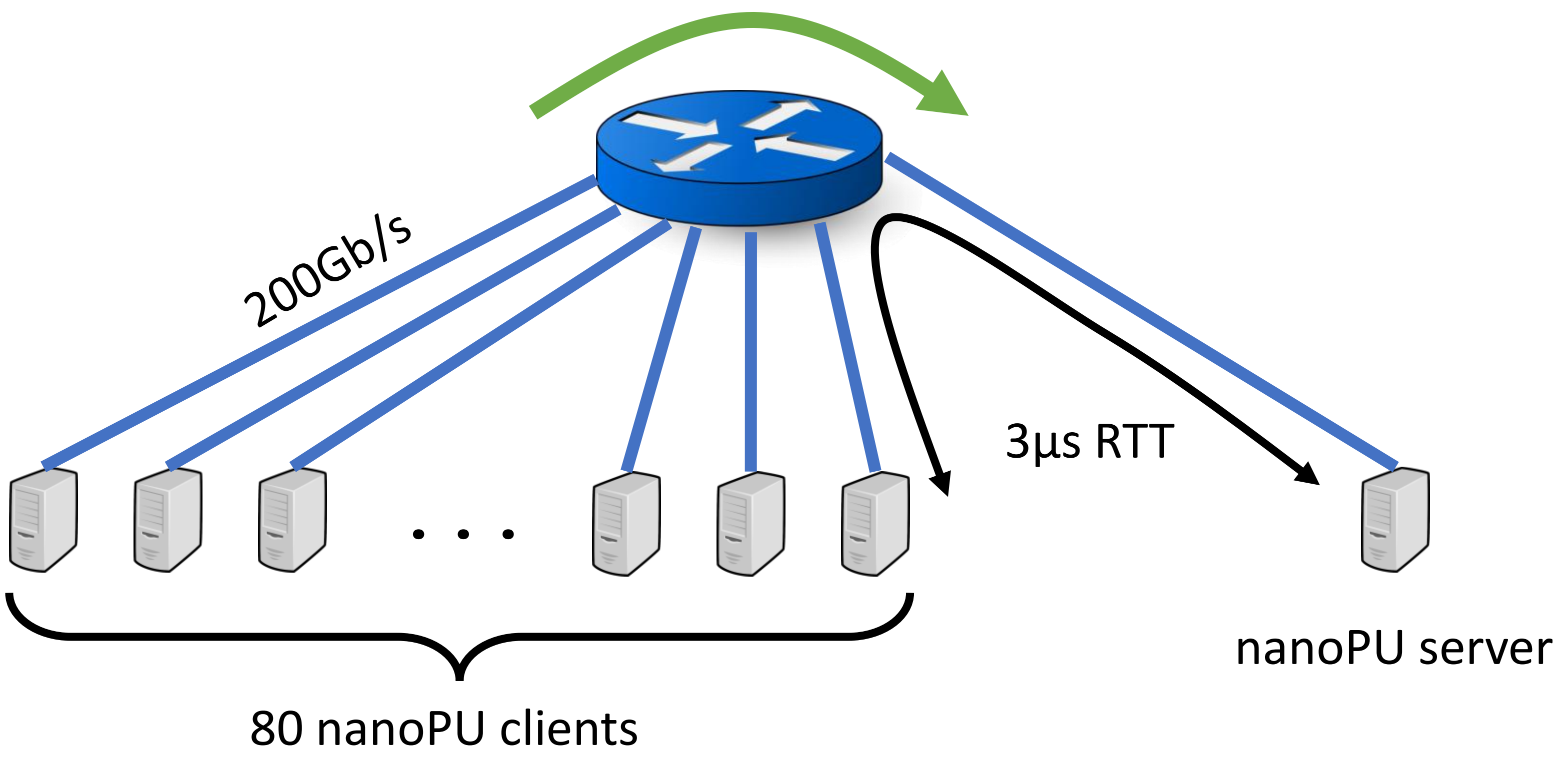
nanoPU Applications

- MICA Key-Value Store:



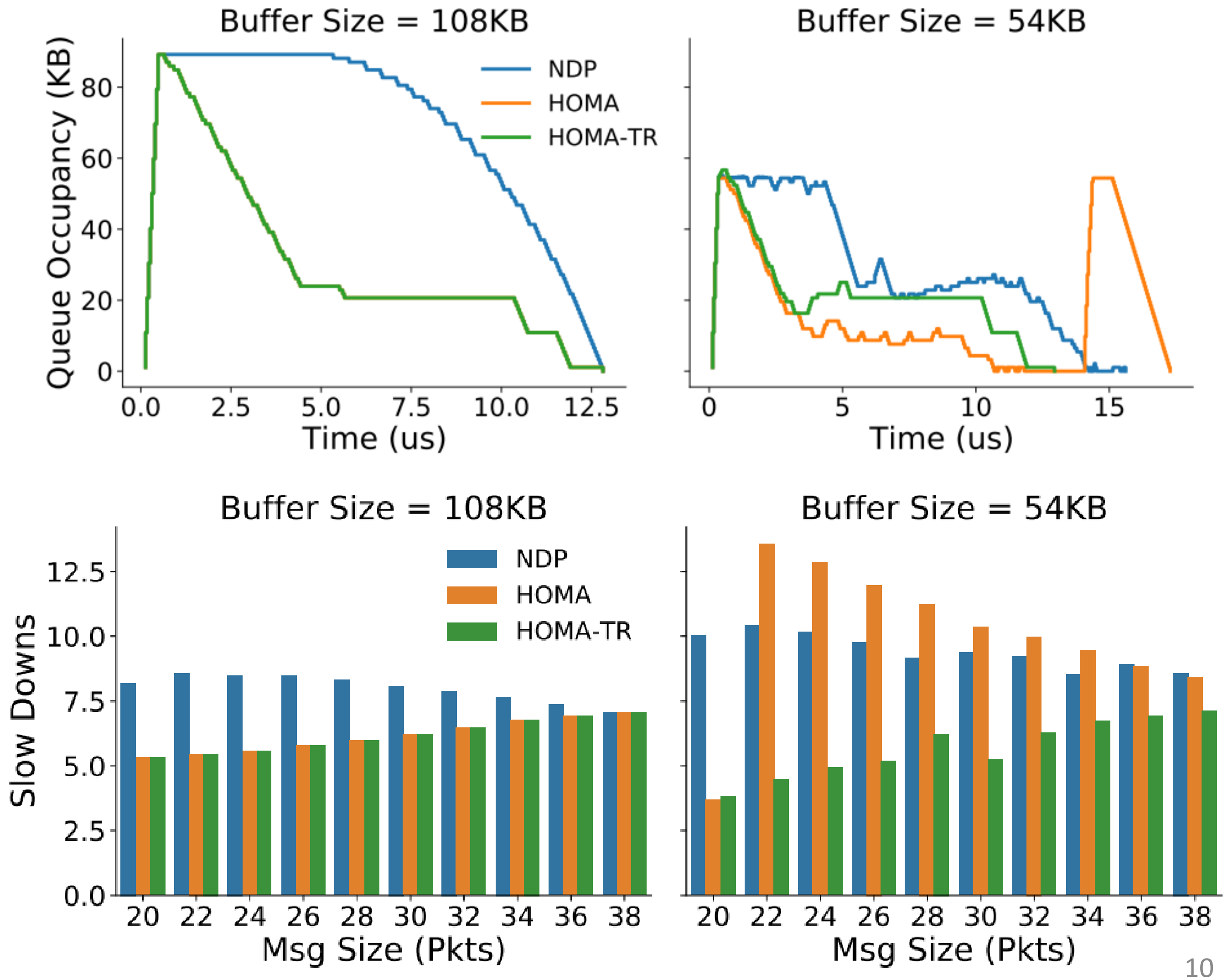
- Raft Consensus, Chain Replication, Set Algebra, and more!

Evaluation Method #3 – Network Simulations



Network Workload Evaluations

- Understand performance characteristics of HW transport protocols
- Run at much larger scale than is possible in grad student lab
- [NanoTransport SOSR '21]



Firesim & HW Transport Protocol Verification

- Many CSPs are exploring and deploying custom transport protocols offloaded to NIC HW
- These protocols are often complex with many tricky edge cases
- Firesim provides an opportunity to:
  - *Validate* correct protocol implementation …
  - *Evaluate* performance of the protocol …
  - Using *real applications* ..
  - *@ large scale* …
  - *Before* silicon tape out
  - Save a lot of *time*, *money*, and *headaches*

nanoSort

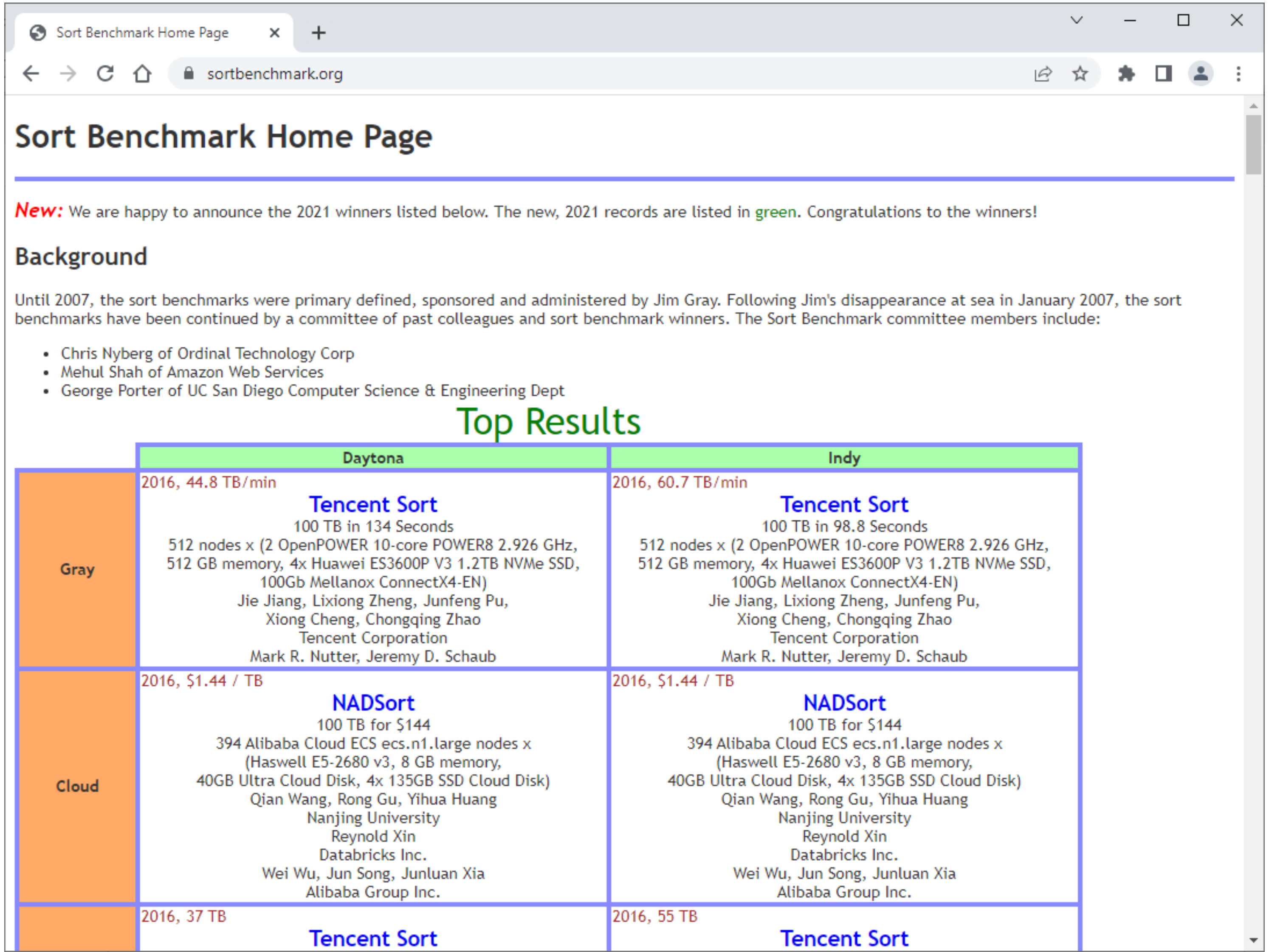
Low-latency, massively parallel sort algorithm using 16,384 cores in FireSim

What?
Sort large datasets as *quickly* as possible.

Why?
Sort is essential for apps with latency deadlines.

How?
New algorithm that leverages low-latency nanoPU communication stack.

GraySort Benchmark



Sort Benchmark Home Page

sortbenchmark.org

Sort Benchmark Home Page

New: We are happy to announce the 2021 winners listed below. The new, 2021 records are listed in green. Congratulations to the winners!

Background

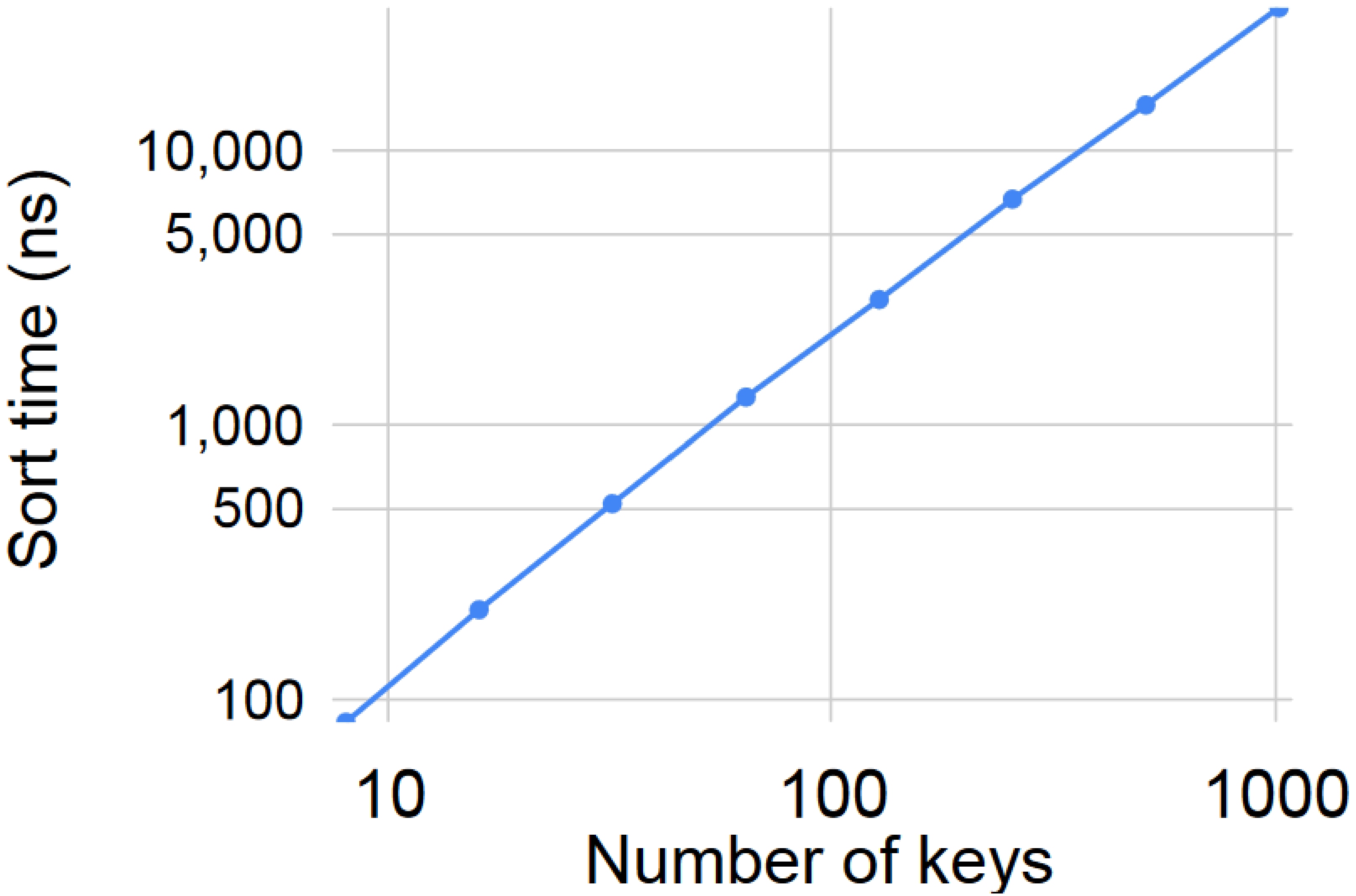
Until 2007, the sort benchmarks were primary defined, sponsored and administered by Jim Gray. Following Jim's disappearance at sea in January 2007, the sort benchmarks have been continued by a committee of past colleagues and sort benchmark winners. The Sort Benchmark committee members include:

- Chris Nyberg of Ordinal Technology Corp
- Mehul Shah of Amazon Web Services
- George Porter of UC San Diego Computer Science & Engineering Dept

Top Results

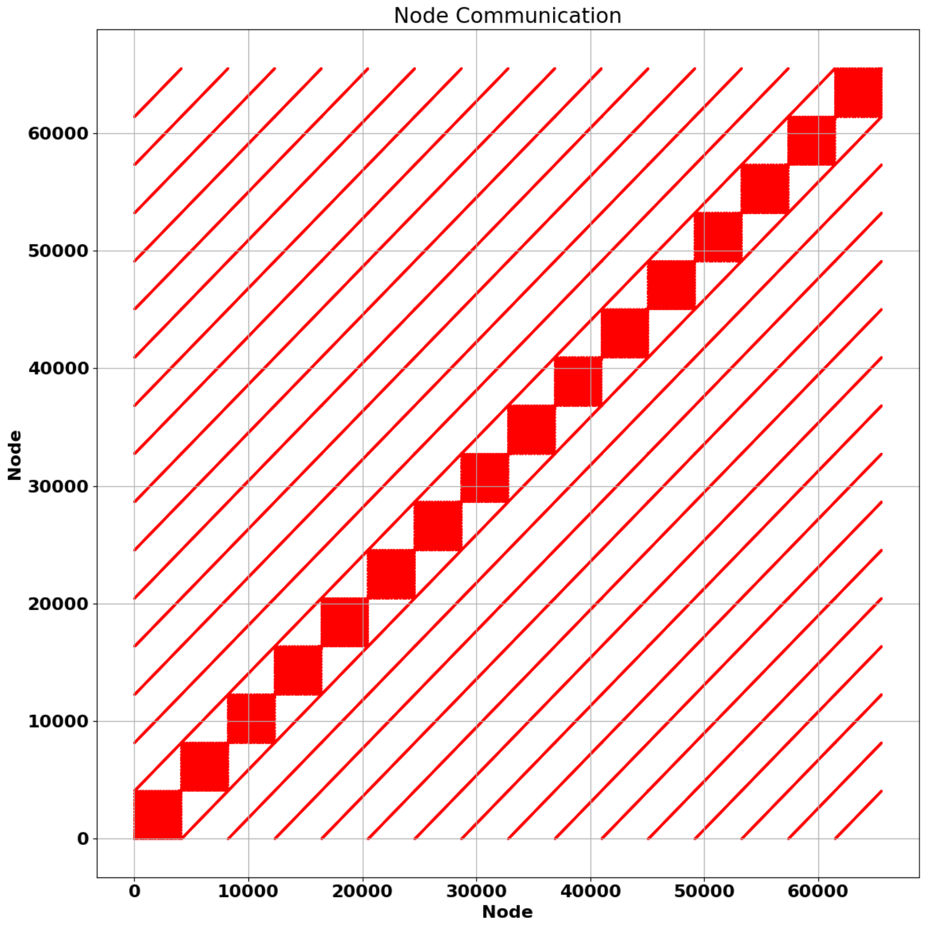
	Daytona	Indy
Gray	2016, 44.8 TB/min **Tencent Sort** 100 TB in 134 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub	2016, 60.7 TB/min **Tencent Sort** 100 TB in 98.8 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub
Cloud	2016, $1.44 / TB **NADSort** 100 TB for $144 394 Alibaba Cloud ECS ecs.n1.large nodes x (Haswell E5-2680 v3, 8 GB memory, 40GB Ultra Cloud Disk, 4x 135GB SSD Cloud Disk) Qian Wang, Rong Gu, Yihua Huang Nanjing University Reynold Xin Databricks Inc. Wei Wu, Jun Song, Junluan Xia Alibaba Group Inc.	2016, $1.44 / TB **NADSort** 100 TB for $144 394 Alibaba Cloud ECS ecs.n1.large nodes x (Haswell E5-2680 v3, 8 GB memory, 40GB Ultra Cloud Disk, 4x 135GB SSD Cloud Disk) Qian Wang, Rong Gu, Yihua Huang Nanjing University Reynold Xin Databricks Inc. Wei Wu, Jun Song, Junluan Xia Alibaba Group Inc.
	2016, 37 TB **Tencent Sort**	2016, 55 TB **Tencent Sort**

Single Core Isn't Fast Enough



The nanoSort Algorithm

1. Shuffle initial keys among all nodes
2. Pick 16 “splitter” keys to delimit buckets
3. Partition nodes into 16 buckets
4. Send keys to nodes in each bucket
5. Recurse in each bucket

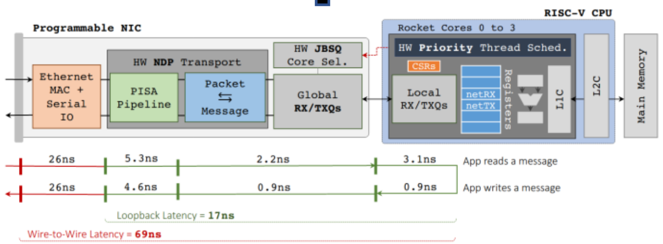


nanoSort Implementation

- nanoSort implemented as a nanoPU C program
- Runs on cluster simulated with FireSim
- Uses 264 AWS EC2 instances (4,224 vCPUs)

nanoSort.c

+



+

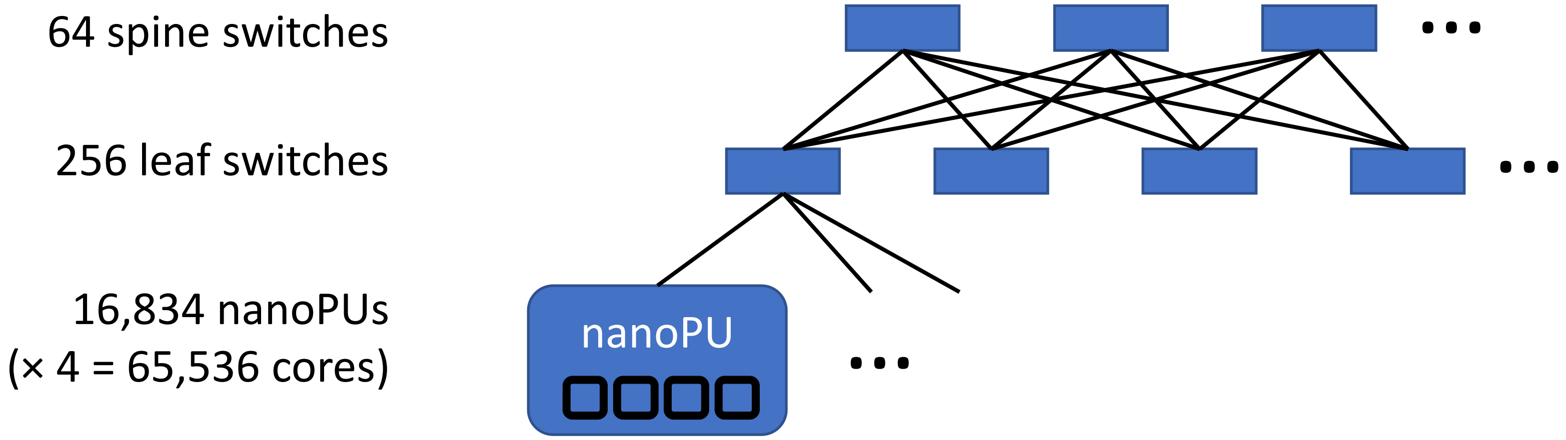
FireSim

+

aws

Network Topology

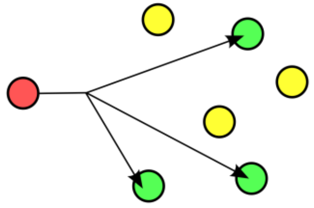
- Full bisection leaf-spine topo with 400G links
- Receiver-driven NDP transport protocol
- In-network support for reliable multicast



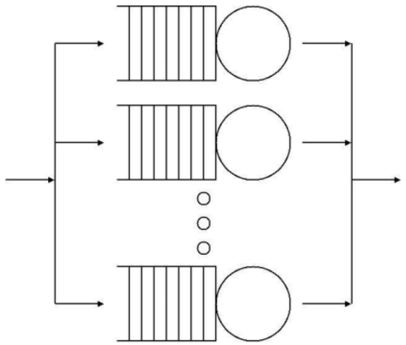
Changes to FireSim

- Support for large scale Verilator simulations
  - Instead of expensive F1 instances, use cheaper c4 instances
  - Not worth FPGA flashing overhead for fast simulations

- Added *reliable* multicast to software switch
  - Switch caches packets in order to handle retransmissions

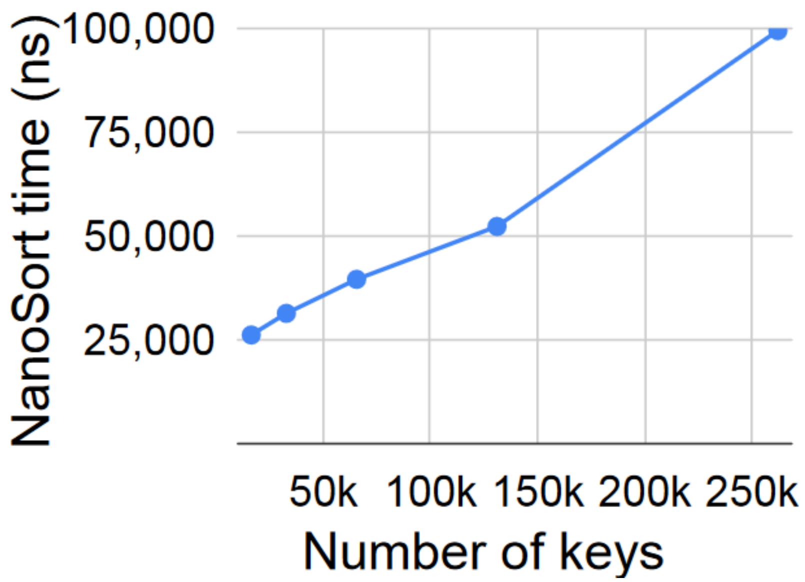


- Changed queueing in software switch
  - Prioritize data over ACK packets

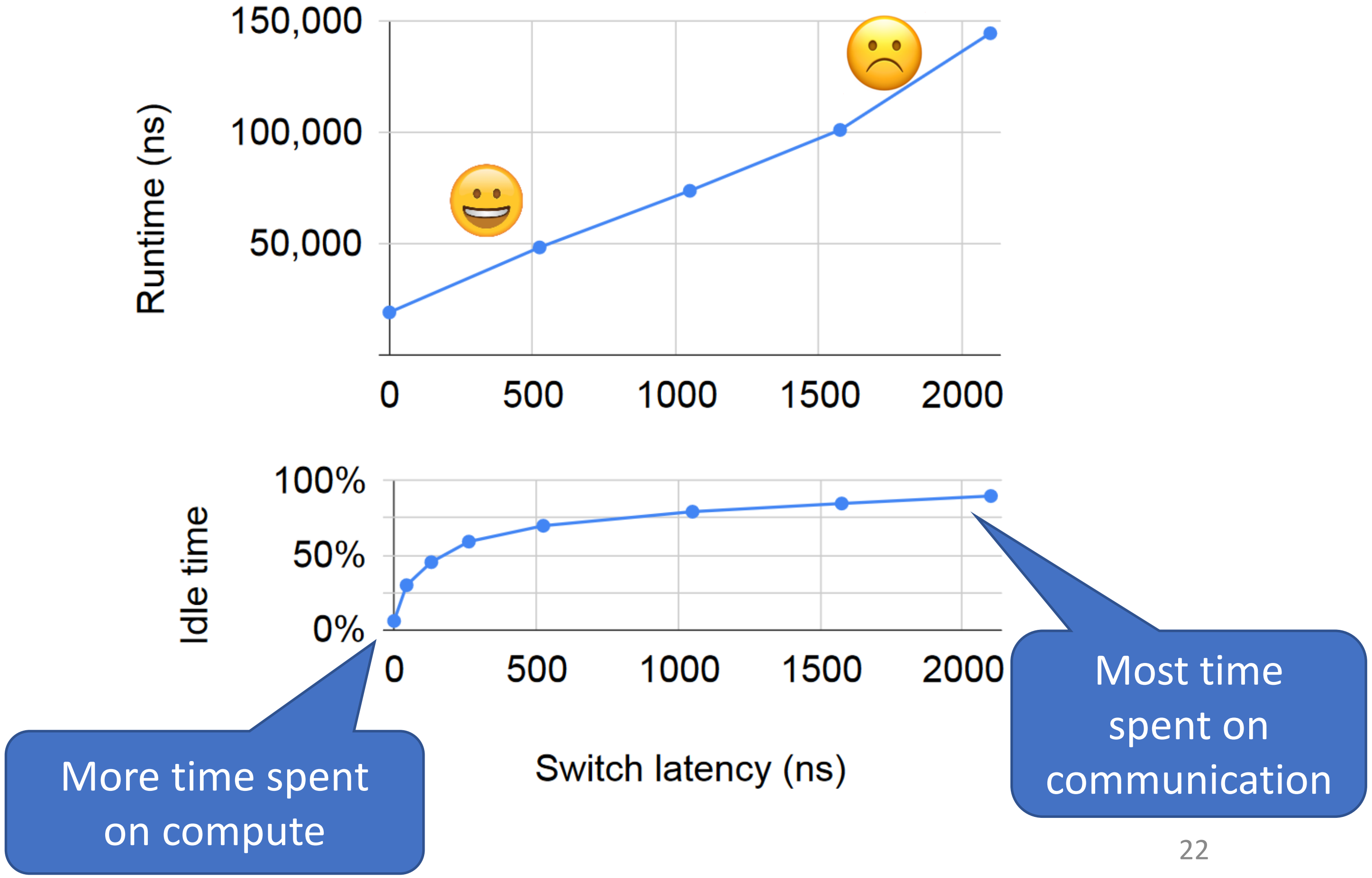


nanoSort Evaluation on FireSim

nanoSort scales



nanoSort needs a fast network

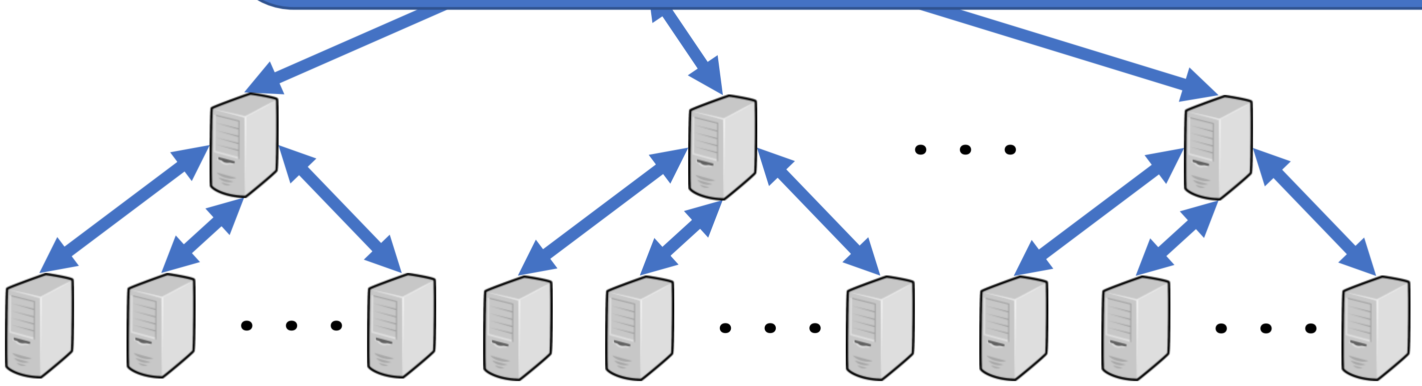


Backup Slides

The Need to Minimize RPC Latency and Software Overheads

Large Online Interactive Services

- Web Search
- Re
- O



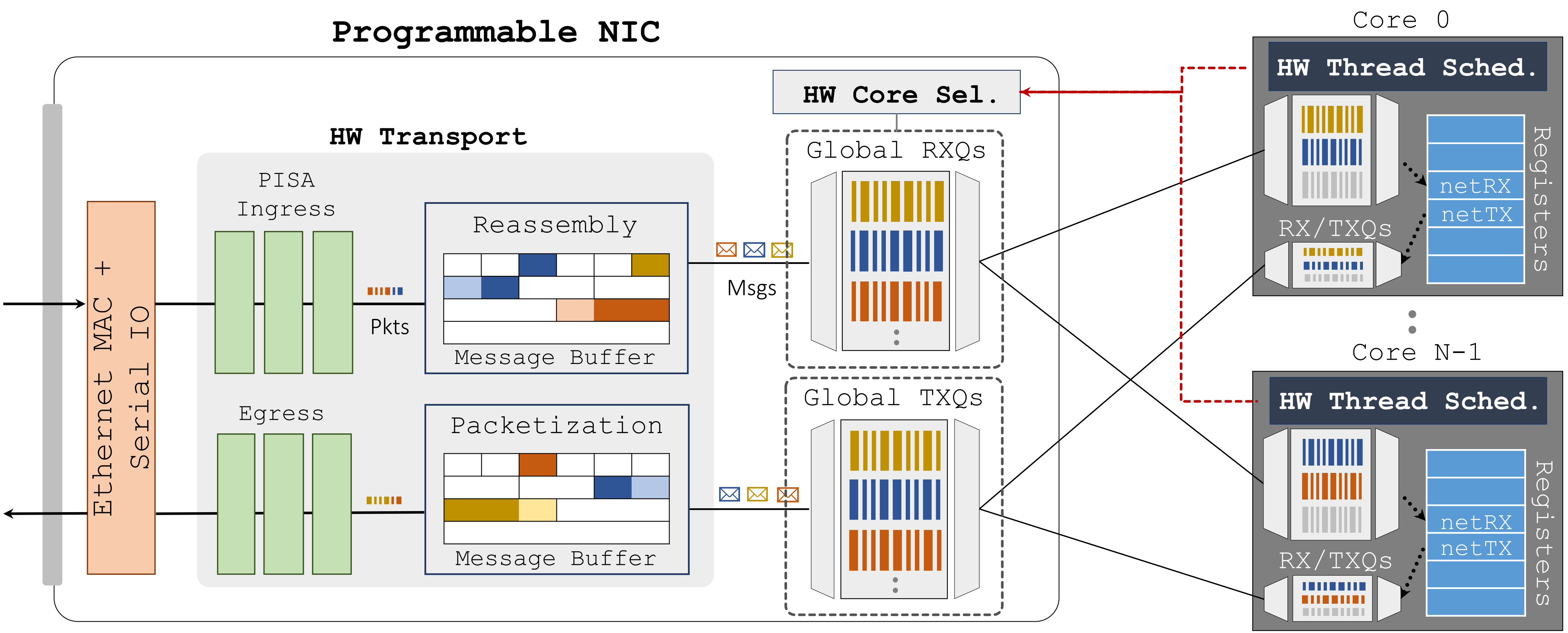
Fine-grained Computing

- Video encoding (ExCamera NSDI'17)
- ocket
- ATC'19)
- cs (Locus
- Flash Bursts (NSDI '21)

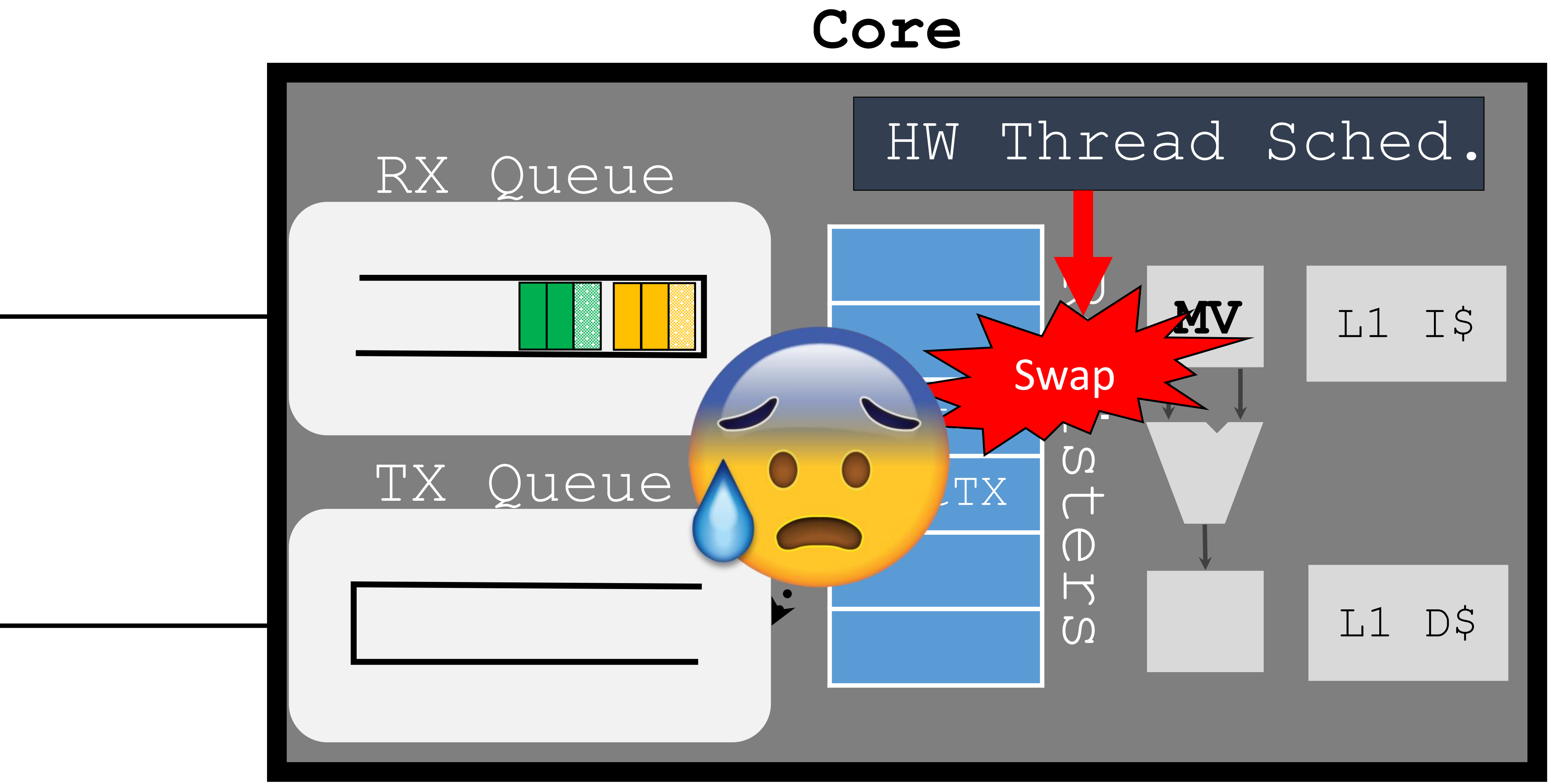
Question:

What would it take to *absolutely minimize* RPC median and tail latency as well as software processing overheads?

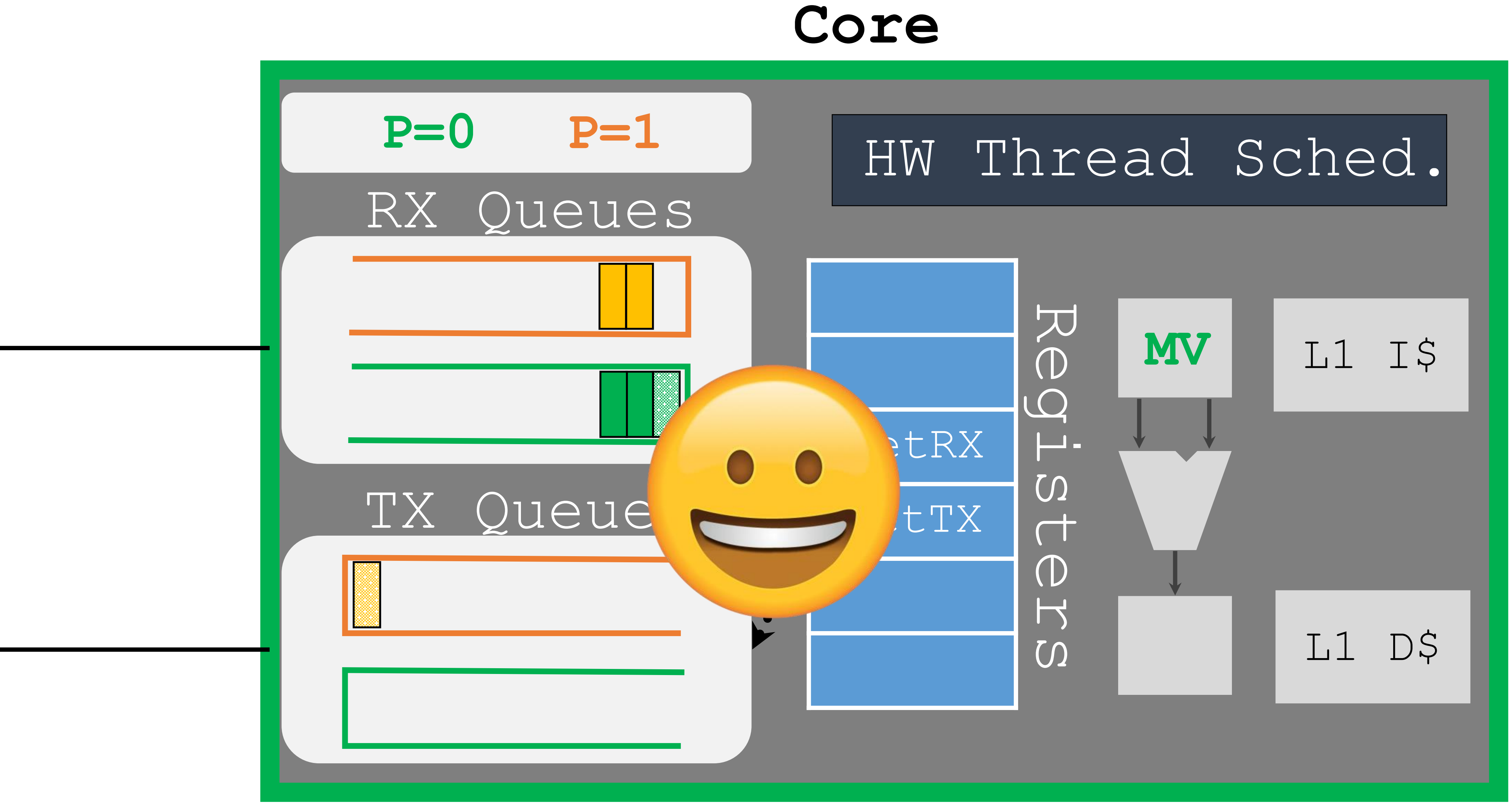
The nanoPU Fast Path



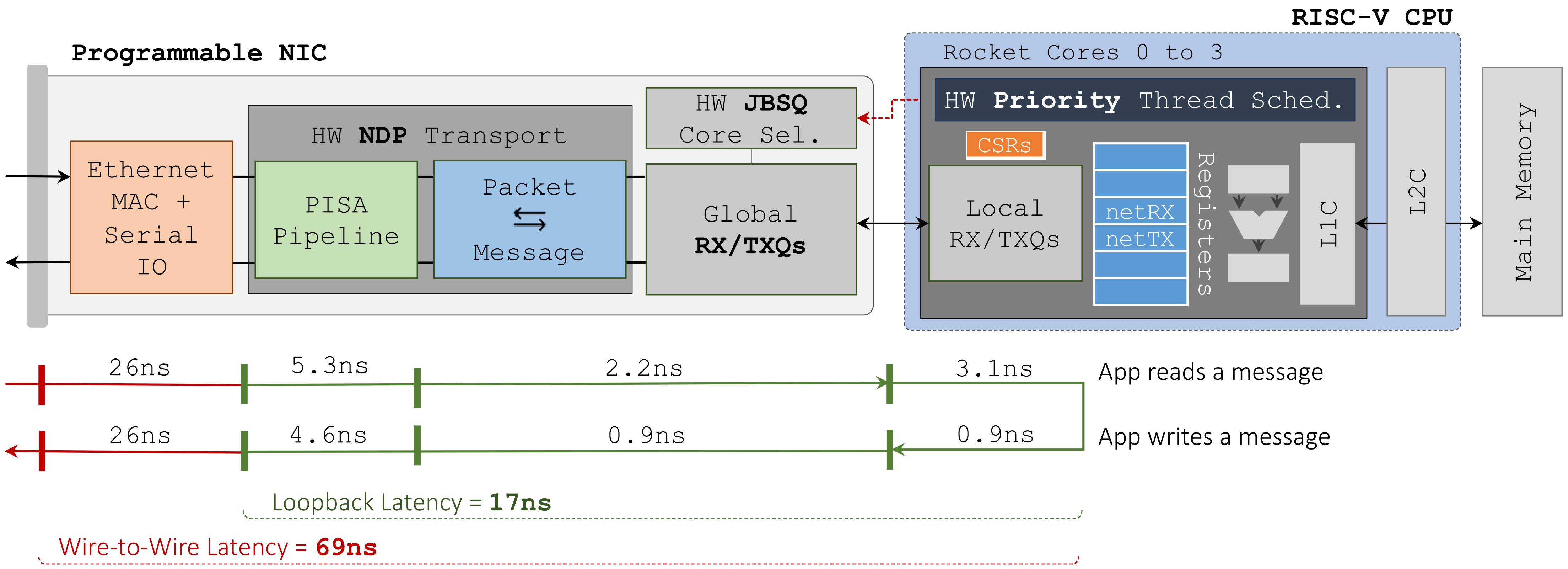
The nanoPU Core



The nanoPU Core



Microbenchmarks



	Wire-to-Wire Latency (ns)	Single Core Loopback Throughput (Mrps)
nanoPU	**69**	**118**
IceNIC	103	16
eRPC	850	10

Evaluation Method #3 – Network Simulations