Wave: Offloading Resource Management to SmartNIC Cores

Jack Tigar Humphries[1]*, Neel Natu[2], Kostis Kaffes[3], Stanko Novaković[2], Paul Turner[2], Henry M. Levy[2,4], David Culler[2,5], Christos Kozyrakis[6]

[1]Stellar Development Foundation
[2]Google Inc
[3]Columbia University
[4]University of Washington
[5]University of California, Berkeley
[6]Stanford University

*Work completed while at Google Inc and Stanford University.

Overview

Why offload system software to a SmartNIC?

Wave design

Closing the host-SmartNIC PCIe latency gap

Evaluation

Why Offload Systems Software to a SmartNIC?

Cloud Efficiency through Offloading

Goal: clouds want to **rent out all CPU cores** and **improve scheduling decisions**

Solution: **offload to SmartNICs** the virtualization & networking tasks and **leverage unique network insight** of SmartNICs

Previous work: offload the OS and networking **data planes**

AWS Nitro, eRSS, AccelNet [NSDI18], Dagger [ASPLOS21], FlexTOE [NSDI22], 1RMA [SIGCOMM20]

New opportunity: also offload OS and networking **system software policies**

Thread scheduling consumes 5% of cycles in Google's fleet [Profiling a warehouse-scale computer]

Why SmartNICs?

Widespread deployment throughout the datacenter

SmartNICs are getting **smarter and more powerful**, more offload opportunities

SmartNIC provides **uniform** security and management platform

Cloud providers can **tailor** SmartNIC design to their workloads

Why Systems Software?

"Systems software" consists of system decision-making tasks

> Kernel thread scheduling, memory management, RPC stack

Only need to invest effort to offload systems software *once* across the fleet

Good fit for SmartNIC cores as decision-making logic is simple

Trusted by cloud providers to run in privileged SmartNIC context

What If We Offload Systems Software Decision Logic?

Architecture: offload policies to the SmartNIC and keep mechanisms on host

Example: Shinjuku scheduling policy logic runs on the SmartNIC [Shinjuku, NSDI 2019]
Interrupt handlers, context switches remain on host

Build upon prior work that already separates OS policy from mechanism

ghOSt [SOSP'21]
Data plane OS's: IX [OSDI 14], Shinjuku [NSDI 19], Snap [SOSP 19], Junction [NSDI 24]
Microkernels: µ-Kernel [SOSP 95], L3, seL4 [SOSP 09], HongMeng [OSDI 24], Zircon
Multikernels: Barrelfish [SOSP 09], NrOS [OSDI 21]
Exokernels: Exokernel [SOSP 95]
Linux userspace subsystems: userfaultfd, libfuse, UIO

Solution: Wave

We **offload three subsystems** to **show that offload is practical**

Wave is a **framework** for offloading system software to the SmartNIC

Builds on **ghOSt's API and kernel-userspace separation**

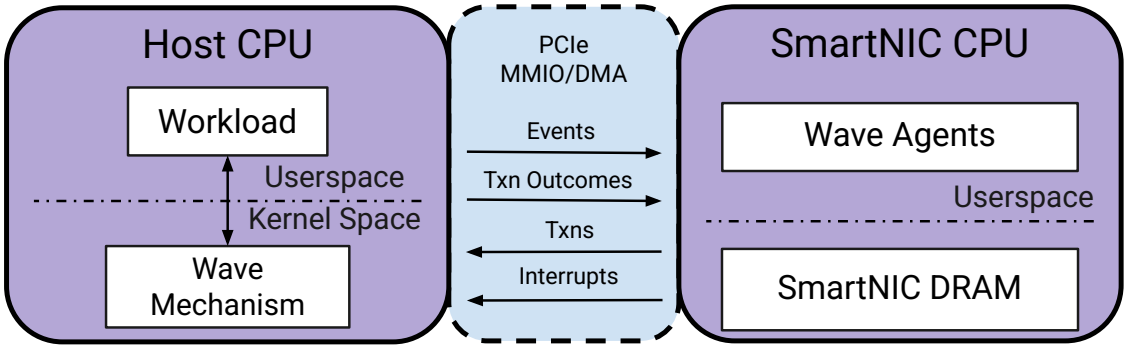
Provides a **general API and mechanism** for host-SmartNIC communication that works well even for *µs-scale* workloads

Wave Design

Wave in a Nutshell

All system software runs in a userspace process on the SmartNIC CPU
Mechanisms along with workloads run on the host CPU

Unidirectional queues for host-SmartNIC communication (Floem, [OSDI 18])
→ Host sends events to SmartNIC agents
→ Agents send decisions back



Intel Mount Evans SmartNIC

ARM Neoverse N1 CPU @ 3.0 GHz with 16 cores

48 GB DRAM

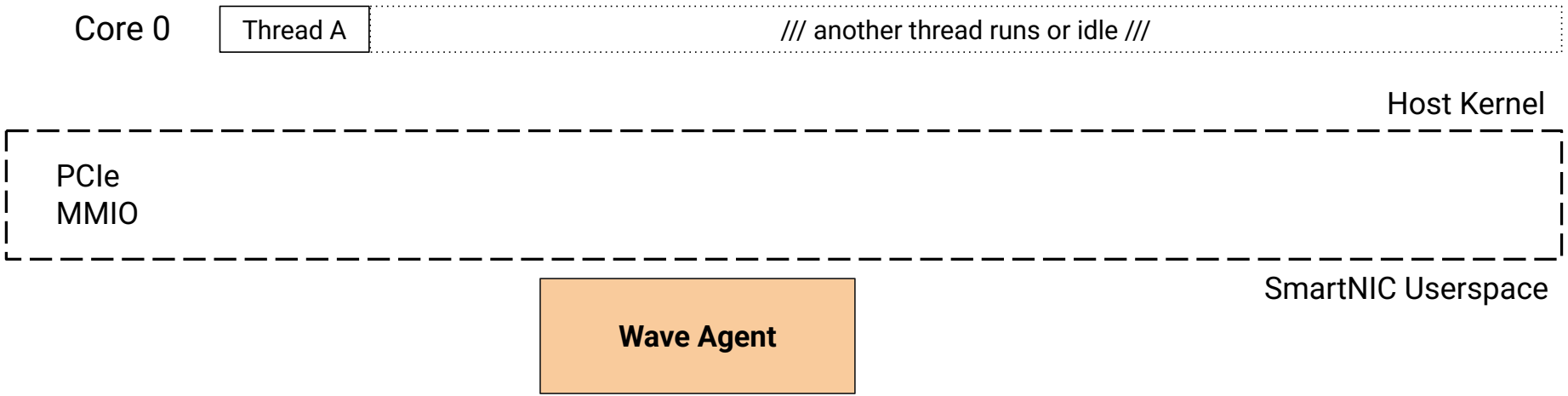
200gbps network throughput

Packet processing and crypto accelerators on-board

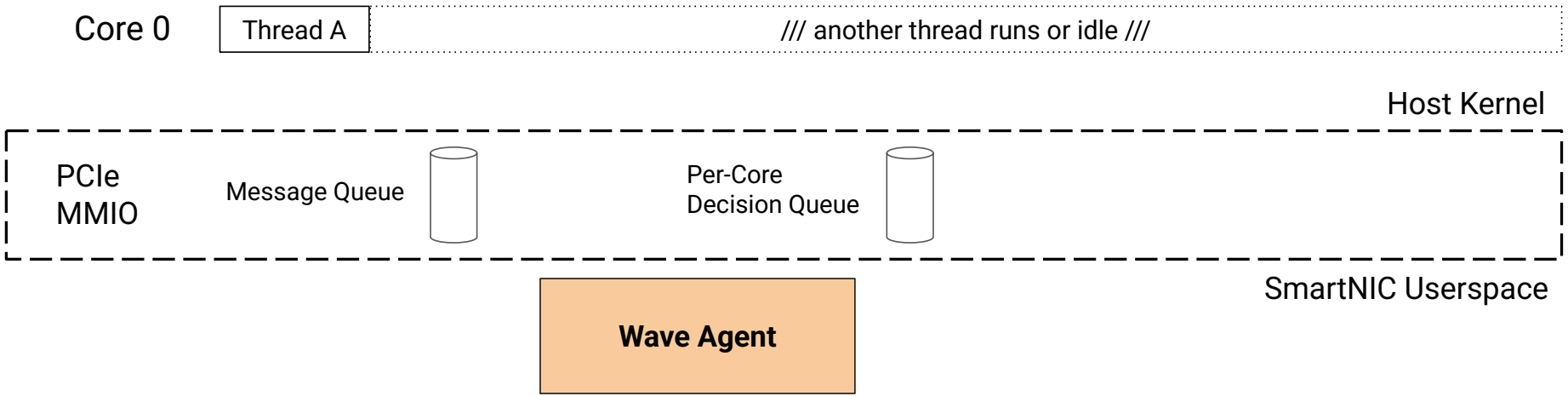
Exposes an MMIO function to the host for reading/writing SmartNIC DRAM

Also has a DMA engine

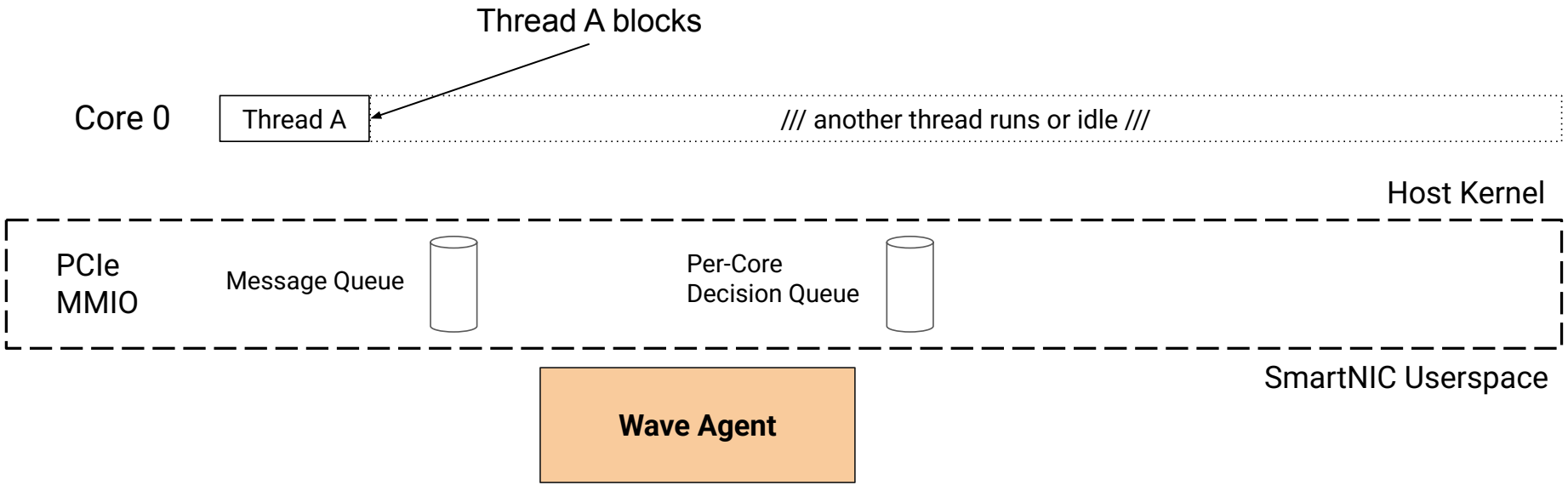
Offloading Scheduling with Wave



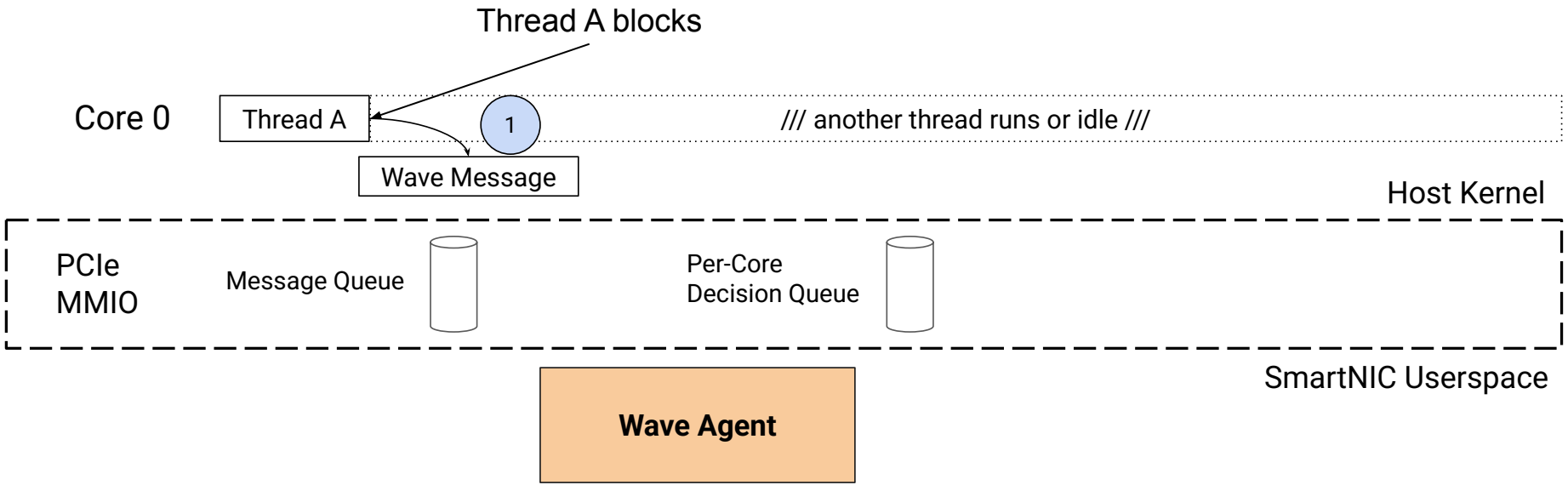
Offloading Scheduling with Wave



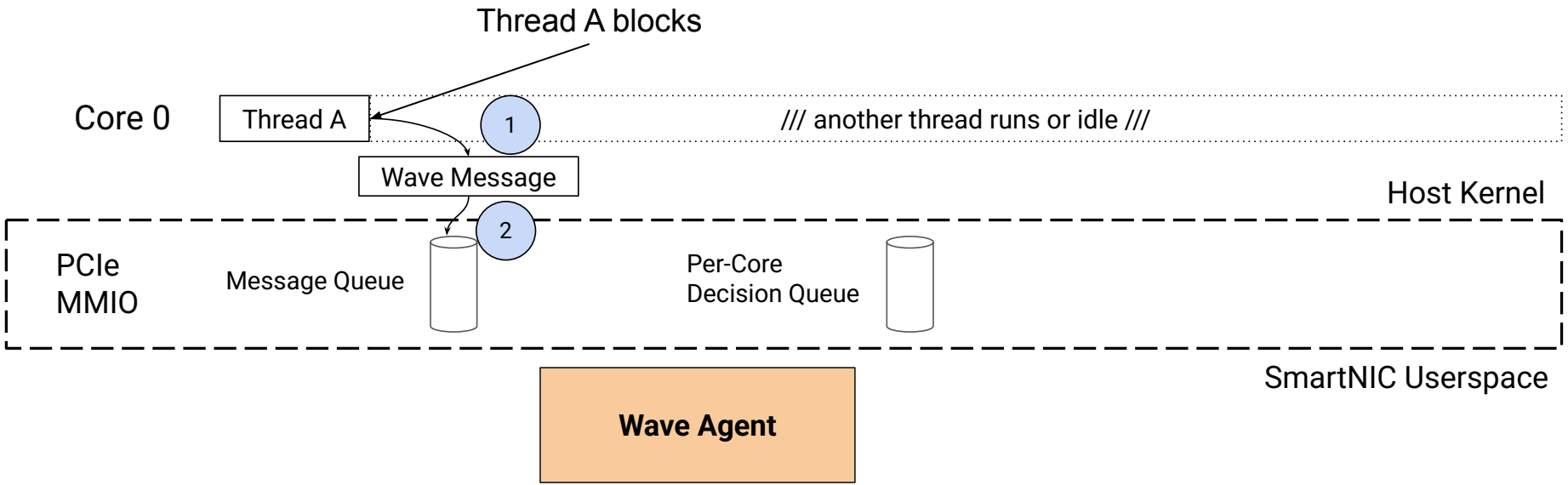
Offloading Scheduling with Wave



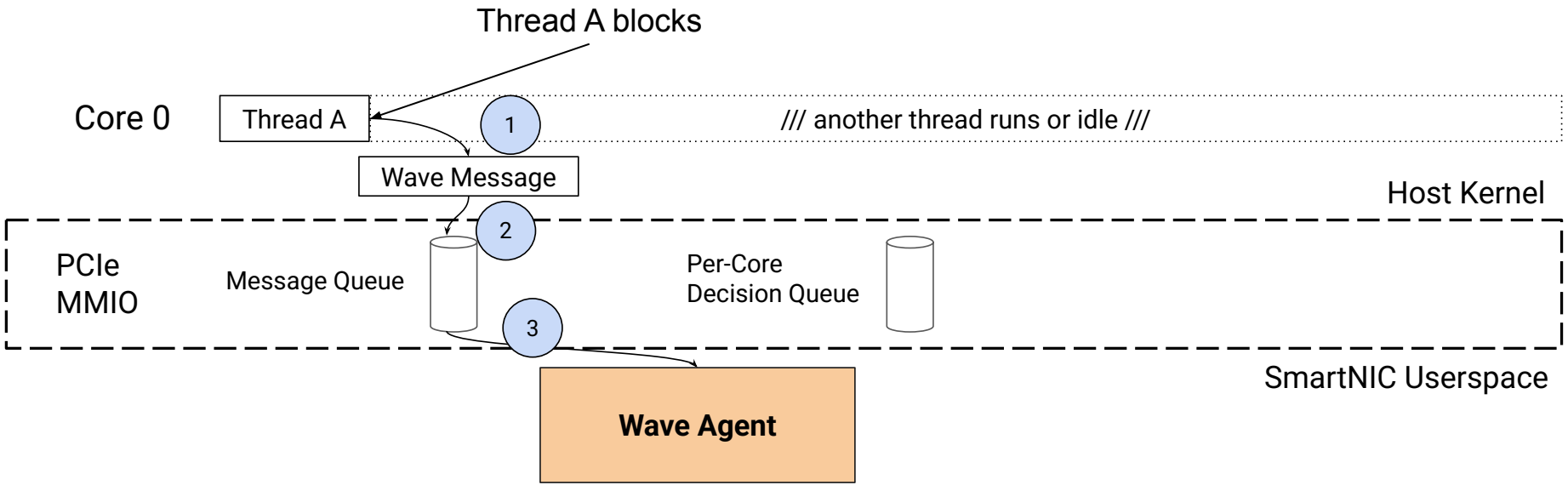
Offloading Scheduling with Wave



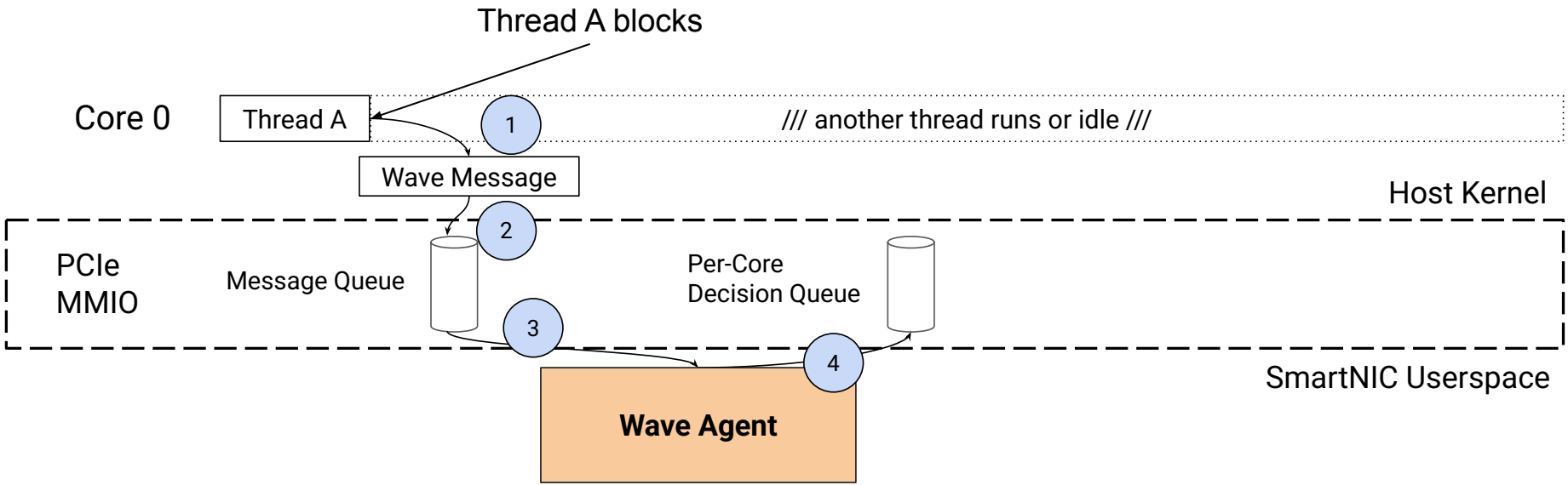
Offloading Scheduling with Wave



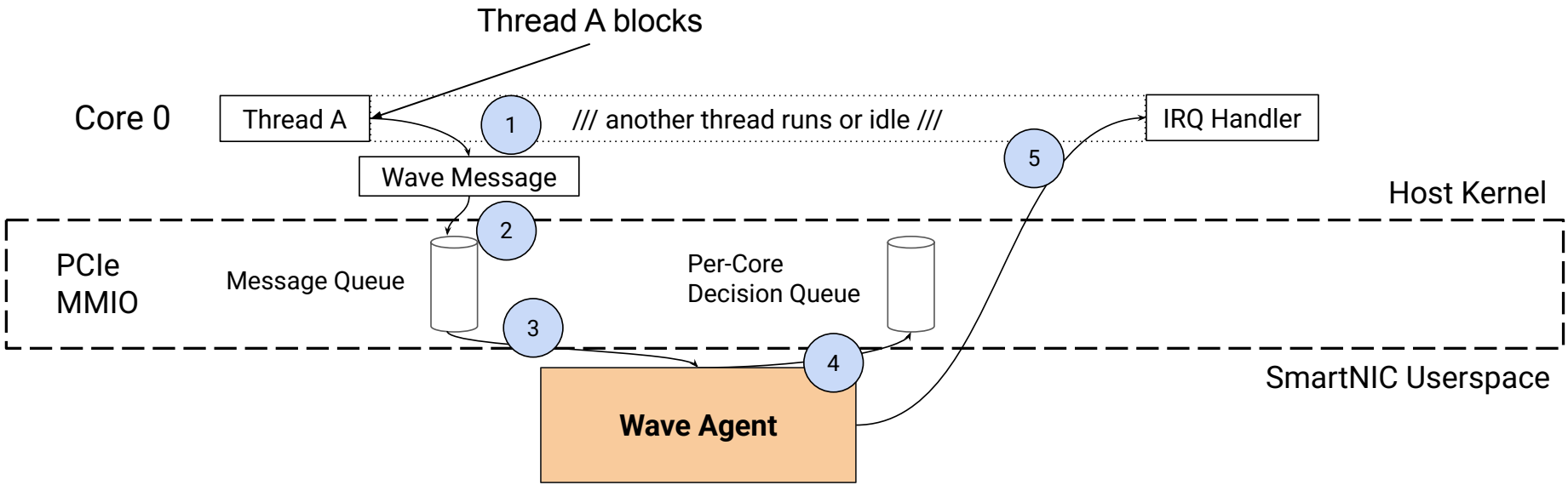
Offloading Scheduling with Wave



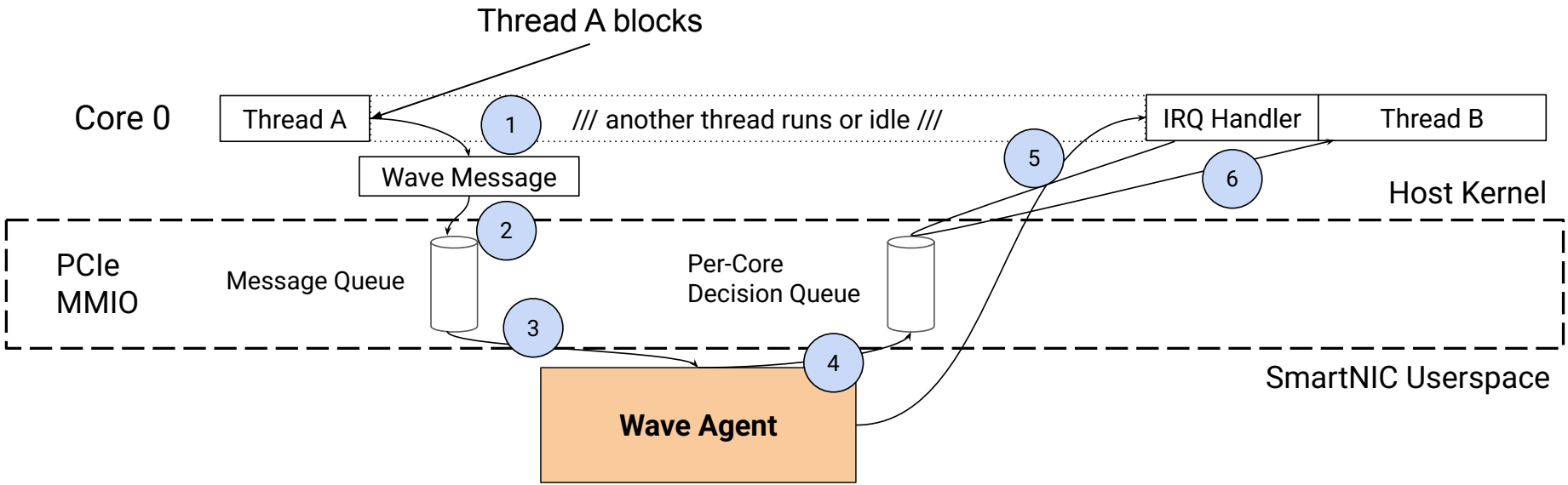
Offloading Scheduling with Wave



Offloading Scheduling with Wave



Offloading Scheduling with Wave



Three Challenges of PCIe

1. **High Latency:** PCIe roundtrip is ~1µs

   Keeps host resources idle; the system performs worse than without the SmartNIC

2. **No coherence** across PCIe bus

3. **No synchronization** scheme for cross-PCIe communication

Closing the Host-SmartNIC Latency Gap

Closing the Host-SmartNIC Latency Gap

Closing the latency gap is *essential* to making offload with Wave practical

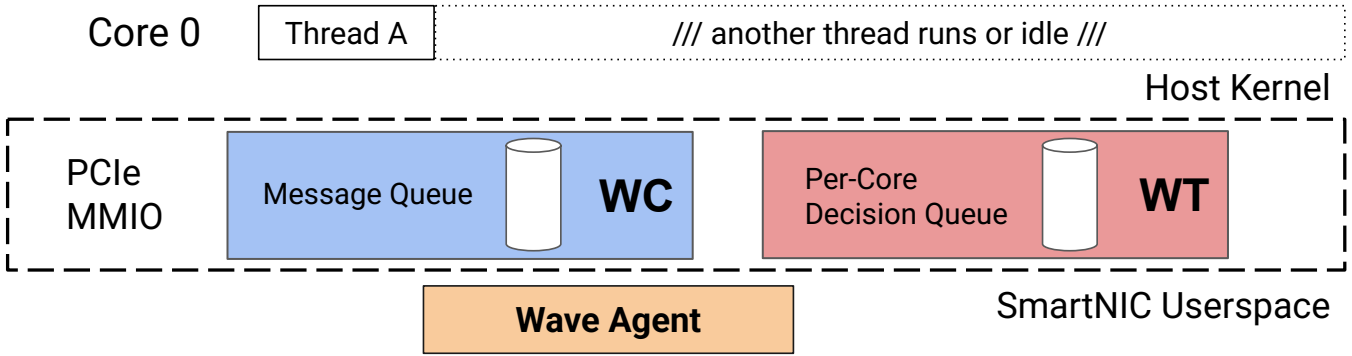
Optimization 1: Tune the page table entry types for MMIO

Optimization 2: Prestage decisions with the agent and prefetch from host

These optimizations improve scheduler throughput by 357%

Tuning MMIO Page Table Entries on the Host

- Map ***message* queue** into host with **write-combining (WC) PTEs**
- Map ***decision* queue** into host with **write-through (WT) PTEs**
  (reads are cached in host cache hierarchy, writes are not cached)
- 31% improvement in scheduler throughput



Prestaging System Software Decisions

The agent can **prestage** decisions in the decision queue

The host can issue a **prefetch** against the queue to hide MMIO latency
WT PTEs allow prefetches against MMIO

Completely **hides** MMIO read cost, improves scheduler throughput by 32%

Evaluation

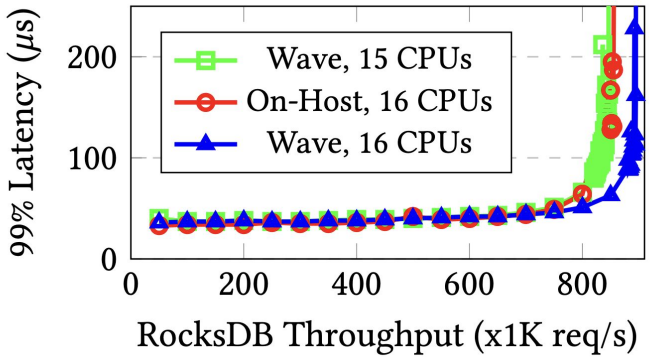
Microbenchmarks: Thread Scheduling

	On-Host with ghOSt	Offload with Wave
Agent creates a decision	770 ns	426 ns
Host context switch overhead	4-5 µs	6-7 µs
…with prestaging and prefetching	2-3 µs	3-4 µs

Wave system software has comparable overheads to on-host software.

Host: AMD Zen3 CPU @ 2.45 GHz with 2 sockets, 64 physical cores per socket, 2 hyperthreads per physical core. Linux 4.15.
SmartNIC: Intel Mount Evans. ARM Neoverse N1 CPU @ 3.0 GHz with 16 physical cores. 200gbps network bandwidth. Linux 5.10.

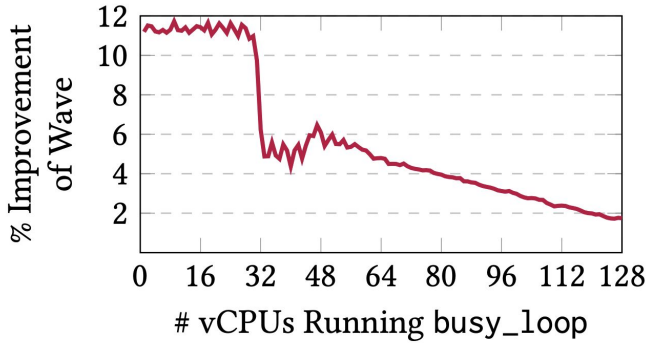
Offloading the FIFO Scheduling Policy



Wave is competitive with on-host scheduling;
Wave improves performance by 4.6% by freeing a host core.

Host: AMD Zen3 CPU @ 2.45 GHz with 2 sockets, 64 physical cores per socket, 2 hyperthreads per physical core. Linux 4.15.
SmartNIC: Intel Mount Evans. ARM Neoverse N1 CPU @ 3.0 GHz with 16 physical cores. 200gbps network bandwidth. Linux 5.10.

Offloading a Virtual Machine Scheduling Policy



Wave improves performance by 2-12% by increasing turbo headroom.

Host: AMD Zen3 CPU @ 2.45 GHz with 2 sockets, 64 physical cores per socket, 2 hyperthreads per physical core. Linux 4.15.
SmartNIC: Intel Mount Evans. ARM Neoverse N1 CPU @ 3.0 GHz with 16 physical cores. 200gbps network bandwidth. Linux 5.10.

Offloading Memory Management

Offload memory management policy to SmartNIC, keep mechanisms on host

Page table entries (PTEs) transferred from host to SmartNIC via **DMA queue**

Memory management agent sends updated PTEs via DMA queue

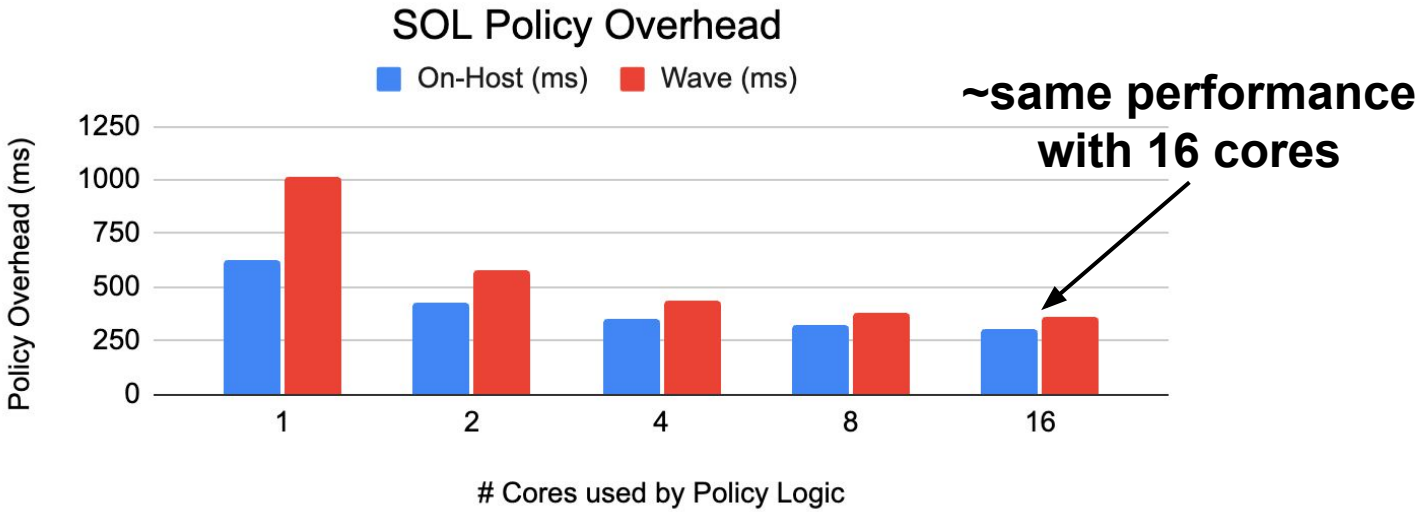
Latency of 100s of milliseconds is acceptable; optimizing for **high throughput**

Offloading Memory Management

We implement the compute-heavy **Safe On-Node Learning (SOL)** memory policy

SOL uses machine learning to cache/evict a 100GiB RocksDB in-memory db

We recover 16 host cores by offloading with Wave



See the full evaluation in the paper.

More microbenchmarks
Shinjuku scheduling policy
RPC stack offload
Coherent interconnect comparison

Summary

Wave shows that system software is a **practical workload** for SmartNICs

Cloud providers **recover host resources** while sacrificing minimal performance

Wave introduces a new **host-SmartNIC API**

Wave optimizes for both **high throughput and low latency** workloads

Wave **closes** the host-SmartNIC PCIe latency gap