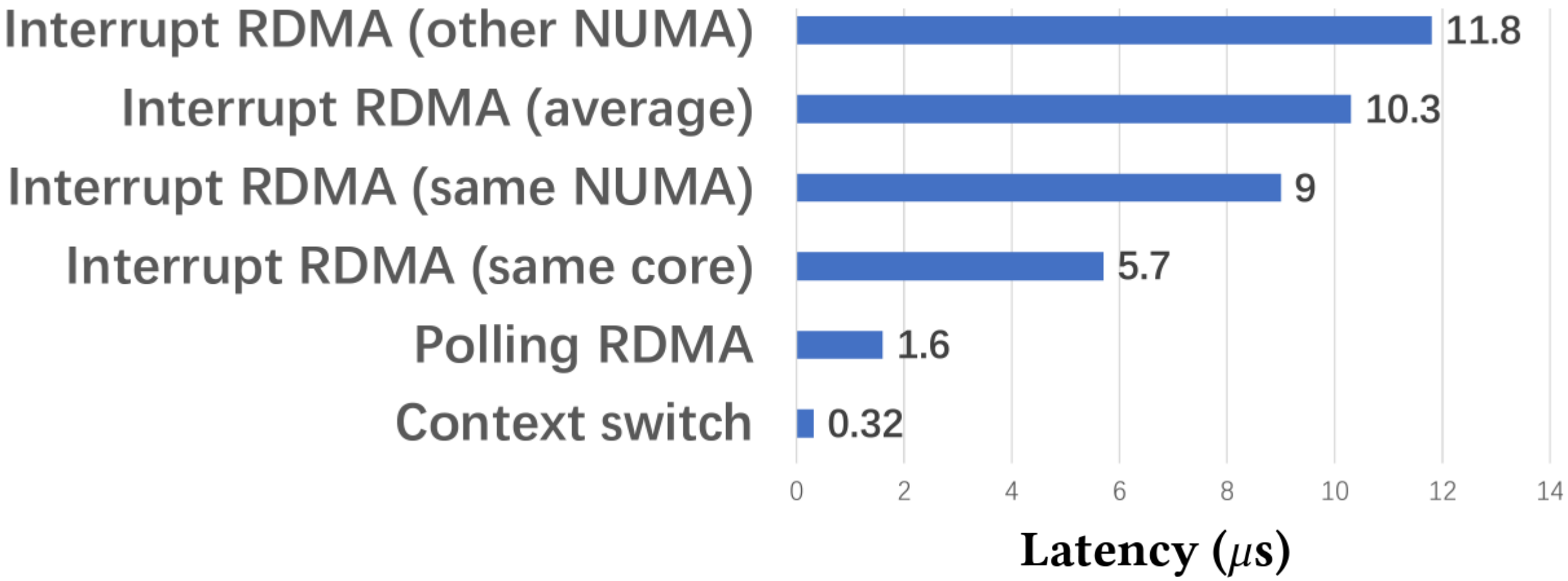
# FastWake: Revisiting Host Network Stack for Interrupt-mode RDMA

Bojie Li[1] Zihao Xiang[1] Xiaoliang Wang[2]
Han Ruan[1] Jingbin Zhou[1] Kun Tan[1]

[1]Huawei [2]Nanjing University

Presenter: Bojie Li, Computer Network and Protocol Lab, Huawei

# Interrupt-mode RDMA wastes the low latency of NICs



# Emergence of Microsecond Events

**Table 1. Events and their latencies showing emergence of a new breed of microsecond events.**

| nanosecond events | microsecond events | millisecond events |
|---|---|---|
| register file: 1ns–5ns | datacenter networking: O(1µs) | disk: O(10ms) |
| cache accesses: 4ns–30ns | new NVM memories: O(1µs) | low-end flash: O(1ms) |
| memory access: 100ns | high-end flash: O(10µs) | wide-area networking: O(10ms) |
| | GPU/accelerator: O(10µs) | |

LUIZ BARROSO, MIKE MARTY, DAVID PATTERSON, AND PARTHASARATHY RANGANATHAN,
Attack of the Killer Microseconds, Communications of the ACM, 2017.

# Why microsecond-scale latencies are hard to hide

## CPU Instruction Pipeline

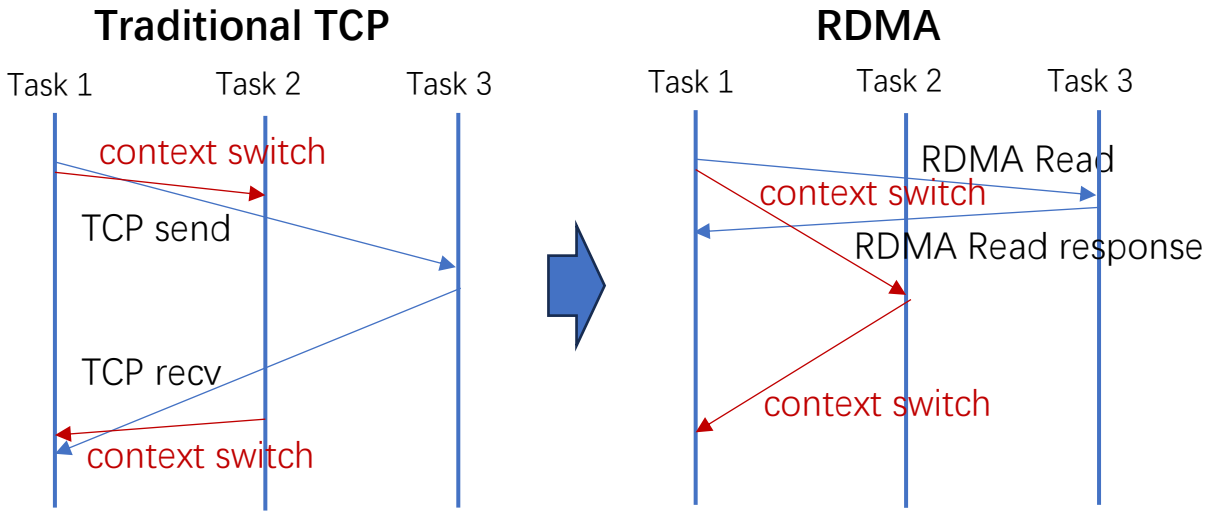
- Nanosecond-scale latencies can be hidden by CPU out-of-order execution pipeline.
- Example: DRAM access takes 50~100 ns, where the CPU core can execute independent instructions after the DRAM access.
- CPU out-of-order pipeline only supports hundreds of instructions, so microsecond-scale latencies would stall the pipeline and decrease the efficiency of CPU.

```
a = *p; // memory access
b = c + d;
e = b + c;
d = b + d;
c = b + c;
f = b + a; // stall for memory access
```

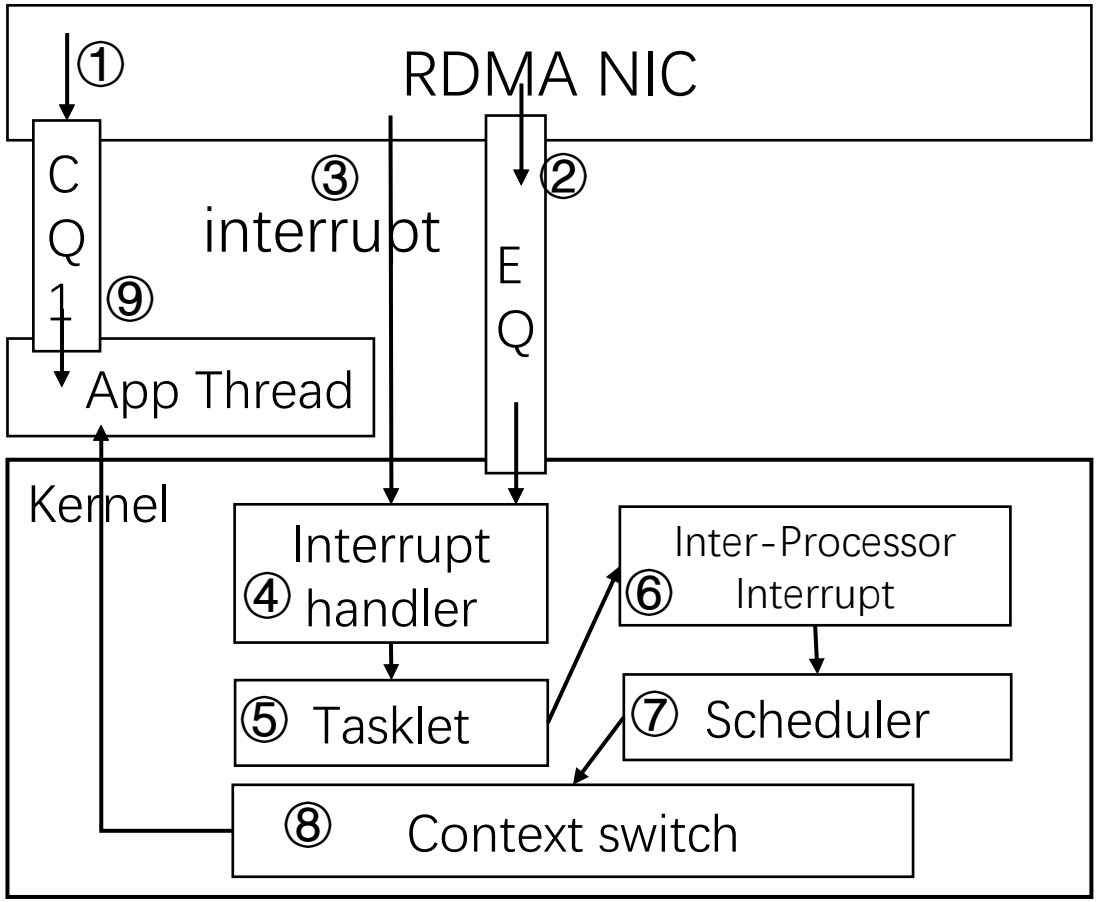
Out-of-order execution

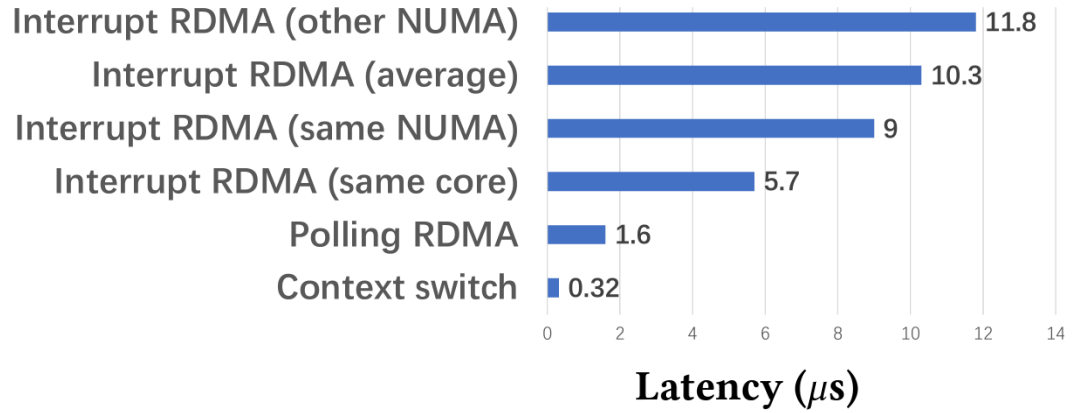
## OS Process Scheduling

- OS process scheduling is designed to hide millisecond-scale latencies.
- Context switching to another process takes 3~5us.
- RDMA Read takes only 2~3us.
- If we context switch to another process after sending the RDMA Read request, and switch back after receiving the RDMA response, then the CPU is wasted on process scheduling.



# Why interrupt-mode RDMA has high latency?



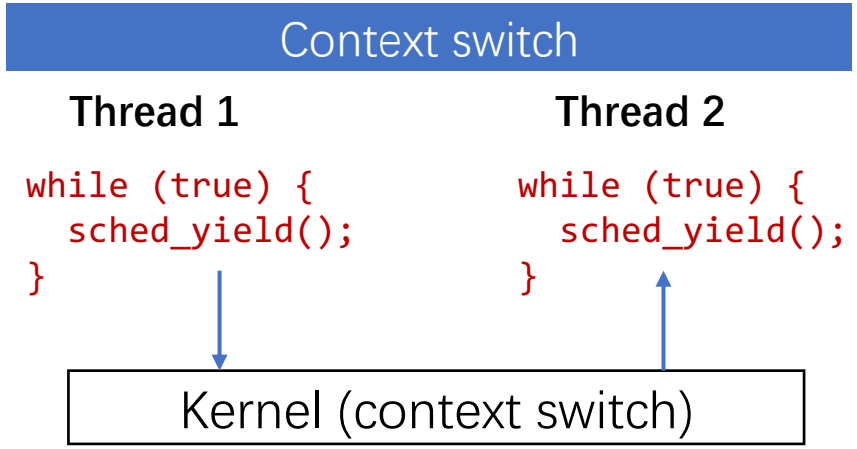



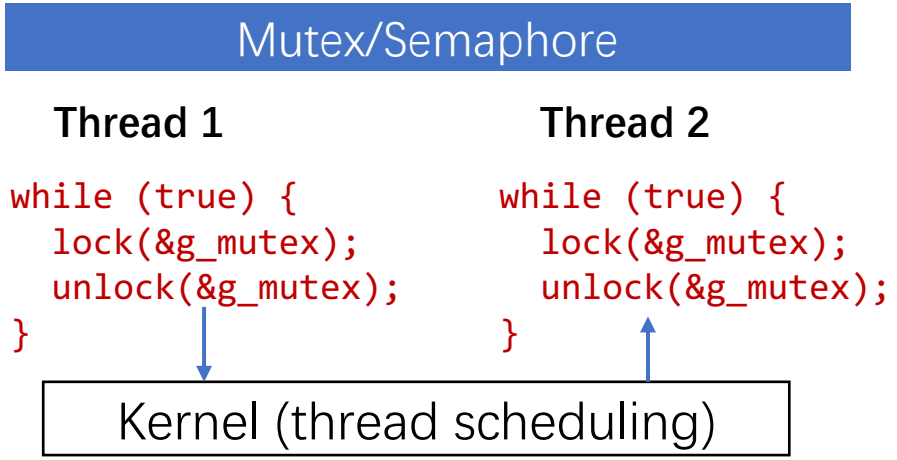

- **CQ: Completion Queue.**
  - Each CQ has a completion vector, which determines the CPU cores to deliver interrupts
- **EQ: Event Queue for interrupt notification.**
  - A completion vector corresponds to an EQ in kernel-mode host memory.
  - EQ entries contain CQ numbers to notify.

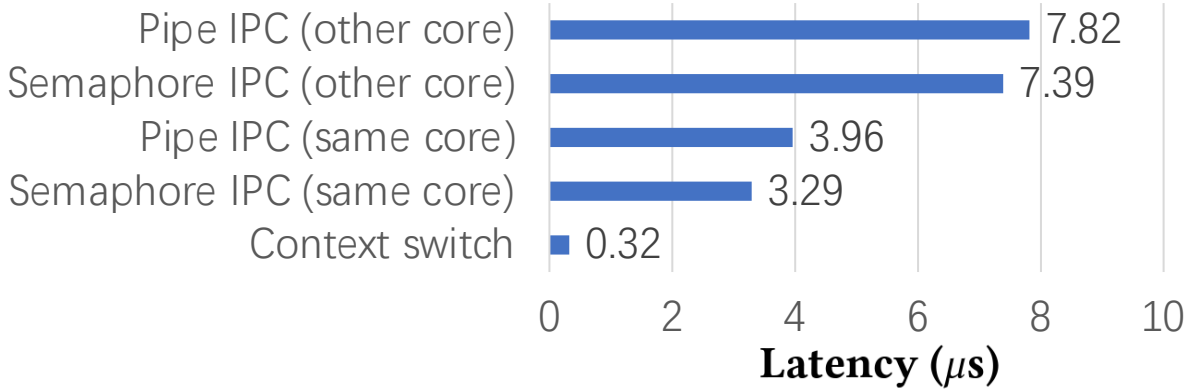
- **①②③ NIC: 1.5 us**
- **④⑤⑦ kernel (same core): 2.2 us**
- **④⑤⑥⑦ kernel (different cores): 6.9 us**
- **⑧ context switch: 0.3 us**

# Key Observation 1

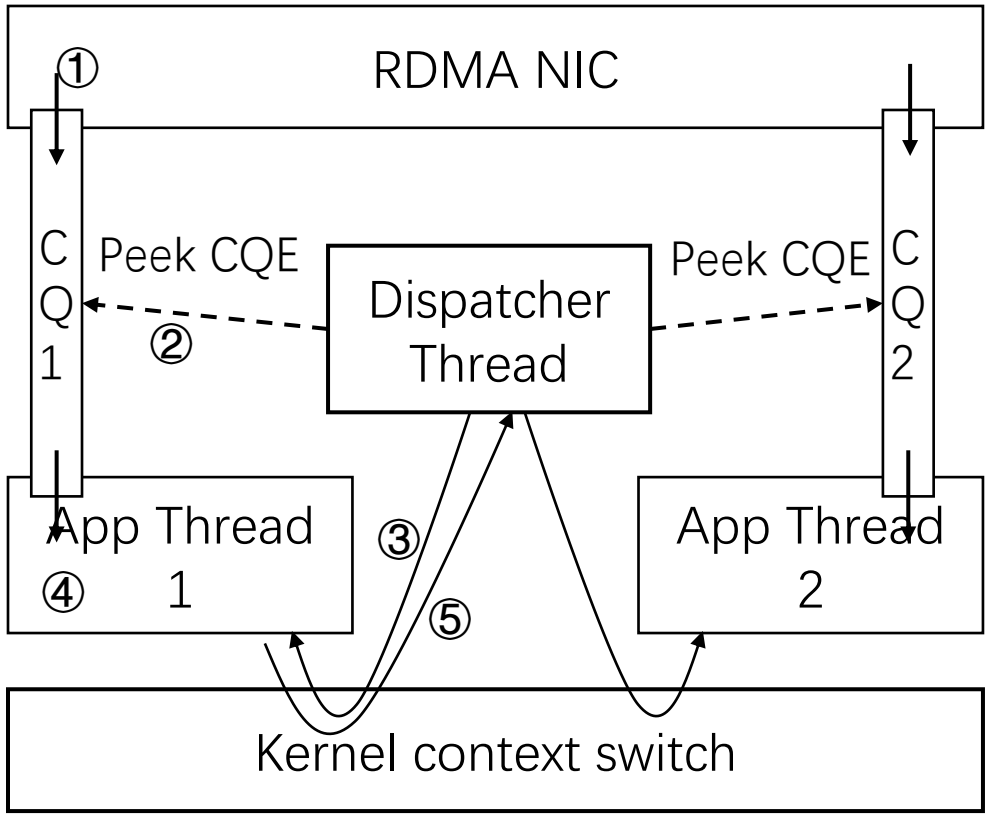
- Direct context switch is much faster than process scheduling.
  - Mutex and semaphore are IPC primitives provided by the kernel that achieves context switch by process scheduling.



# Approach 1: per-core dispatcher thread




The CQ is mapped to both application thread and a per-core dispatcher thread.

1. NIC sends CQE to the CQ.
2. Dispatcher thread polls all CQs on the core and peeks CQE from the CQ, but do not pop it out of CQ.
3. **Dispatcher thread context switches to the application thread directly.**
4. The application thread pops the CQE out of CQ.
5. When the application waits on the next event, it context switches back to the dispatcher thread.

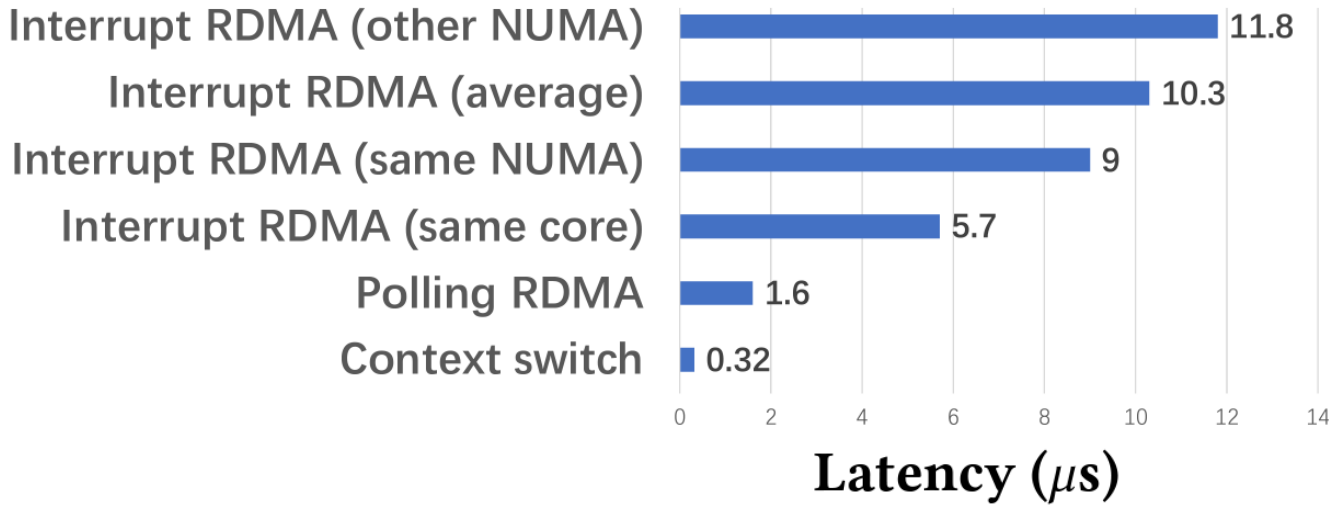
# How to implement direct context switch?

- Linux does not support context switching to a specific thread.
- We introduce a new system call `switch_to(pid)`
  - Simply checks the permission and puts the thread `pid` to the head of runqueue.
  - Performance: 0.3~0.4 us.

- Security:
  - To avoid switch_to(pid) being abused to starve other threads, we only allow non-dispatcher threads to switch to dispatcher threads and only allow dispatcher threads to switch to non-dispatcher threads.
    - We introduce a flag bit in the process control block to indicate whether it is a dispatcher thread.
- Limitations:
  - Direct context switch makes application threads have high priority than other threads with the same priority. However, because each priority has its own runqueue, threads with higher priority still takes precedence.

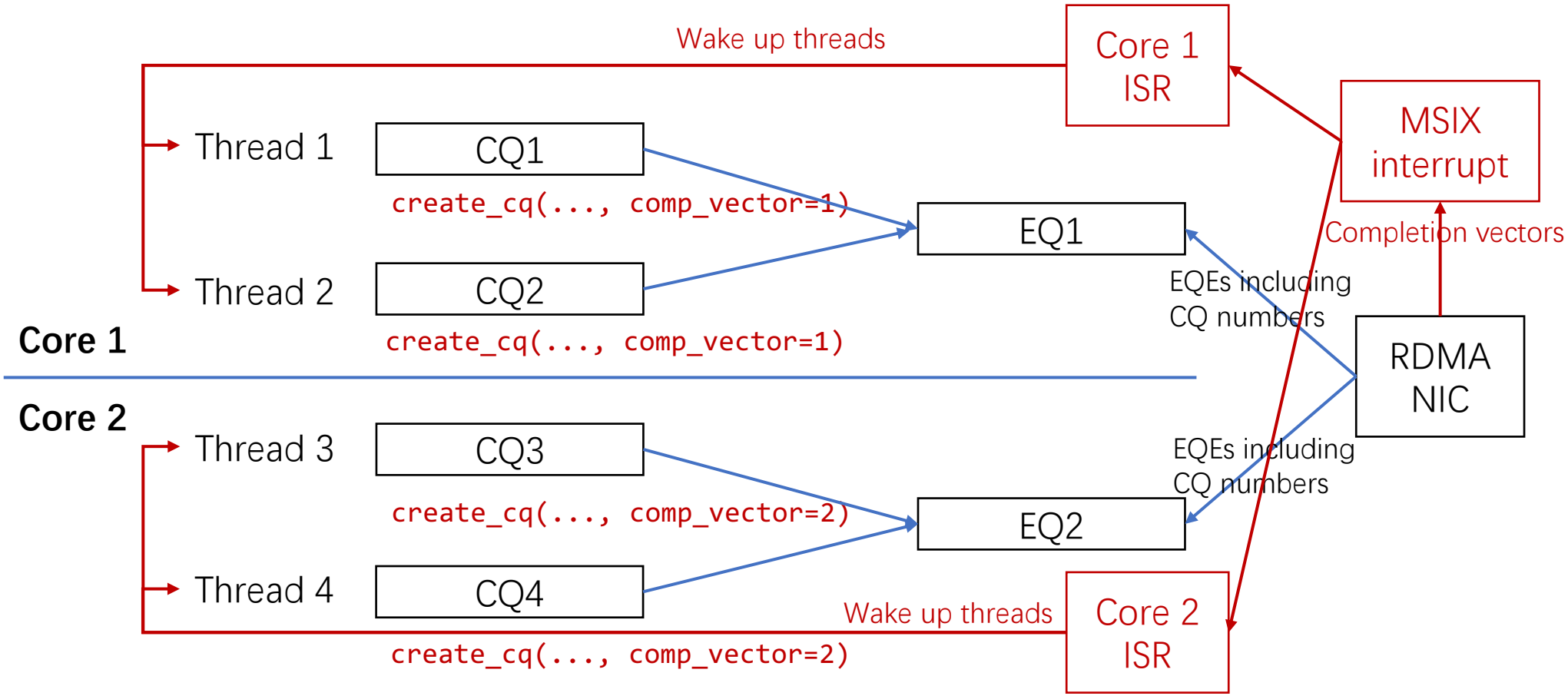
# Key Observation 2

- Interrupt delivery to a thread running on the same core is much faster than other cores.
- Reason: inter-processor interrupts (IPI) to wake up another core.



# Approach 2: interrupt core affinity & shorten kernel path

- How to make sure interrupts and the thread are on the same core?



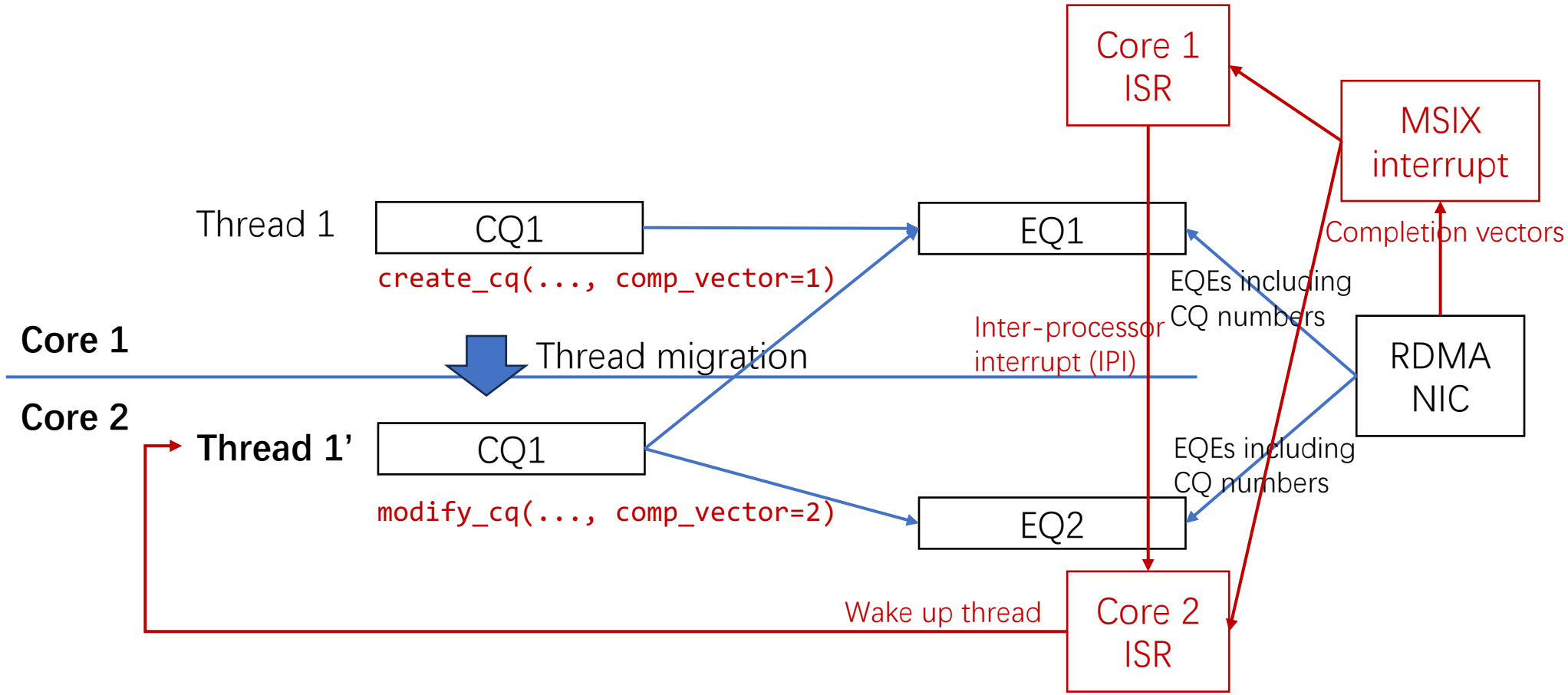
# Approach 2: interrupt core affinity & shorten kernel path

- What if the creator of the CQ and the user of the CQ are in different threads and on different cores?
- What if a thread migrates to another core?
- We need to dynamically update the affinity between CQs and EQs.

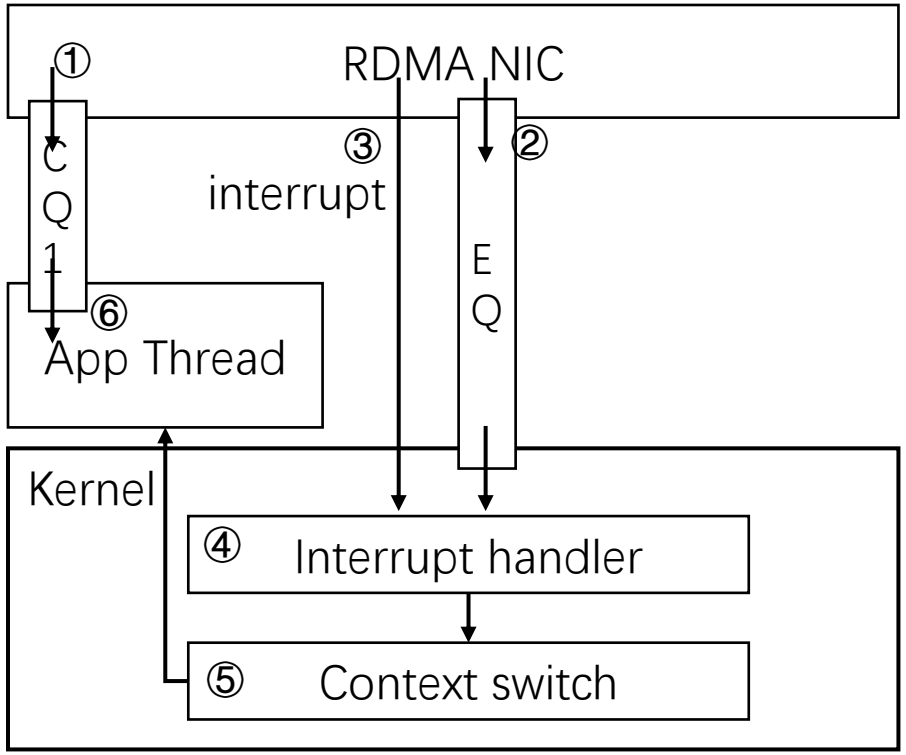
- We leverage a feature in Mellanox NICs: **CQ-to-EQ remapping**.
- The remapping is done lazily when the interrupt handler finds the thread to wake up is not on the same core.

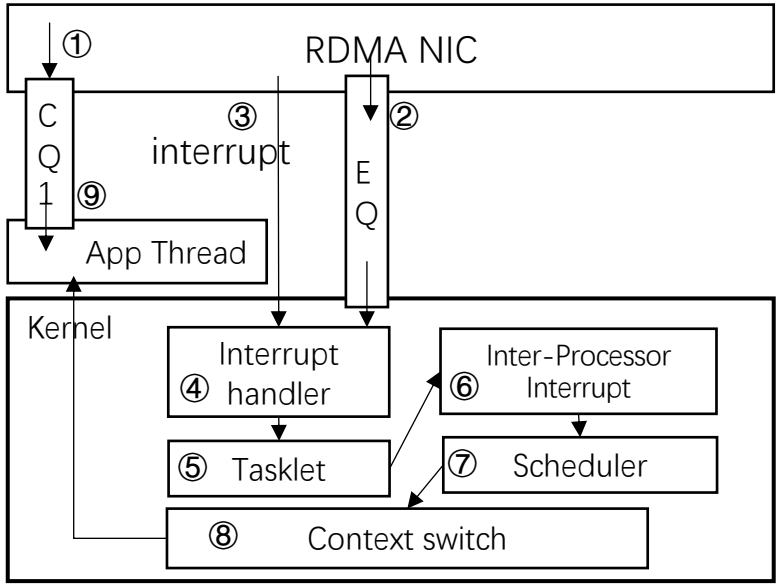
# Approach 2: interrupt core affinity & shorten kernel path

- Do CQ-to-EQ remapping when interrupt and thread are on different cores.



# Approach 2: interrupt core affinity & shorten kernel path

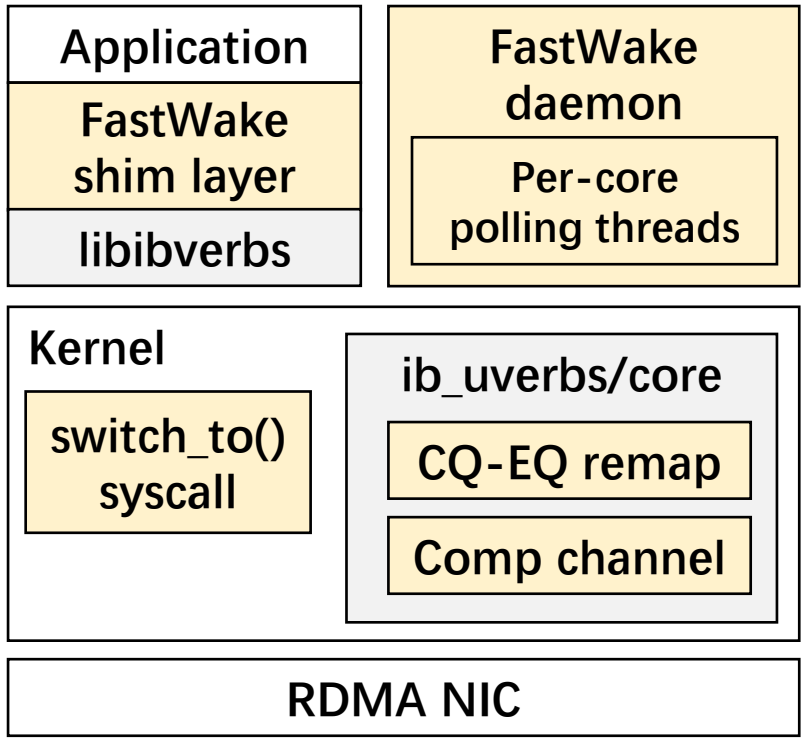



Compared to traditional interrupt delivery path:

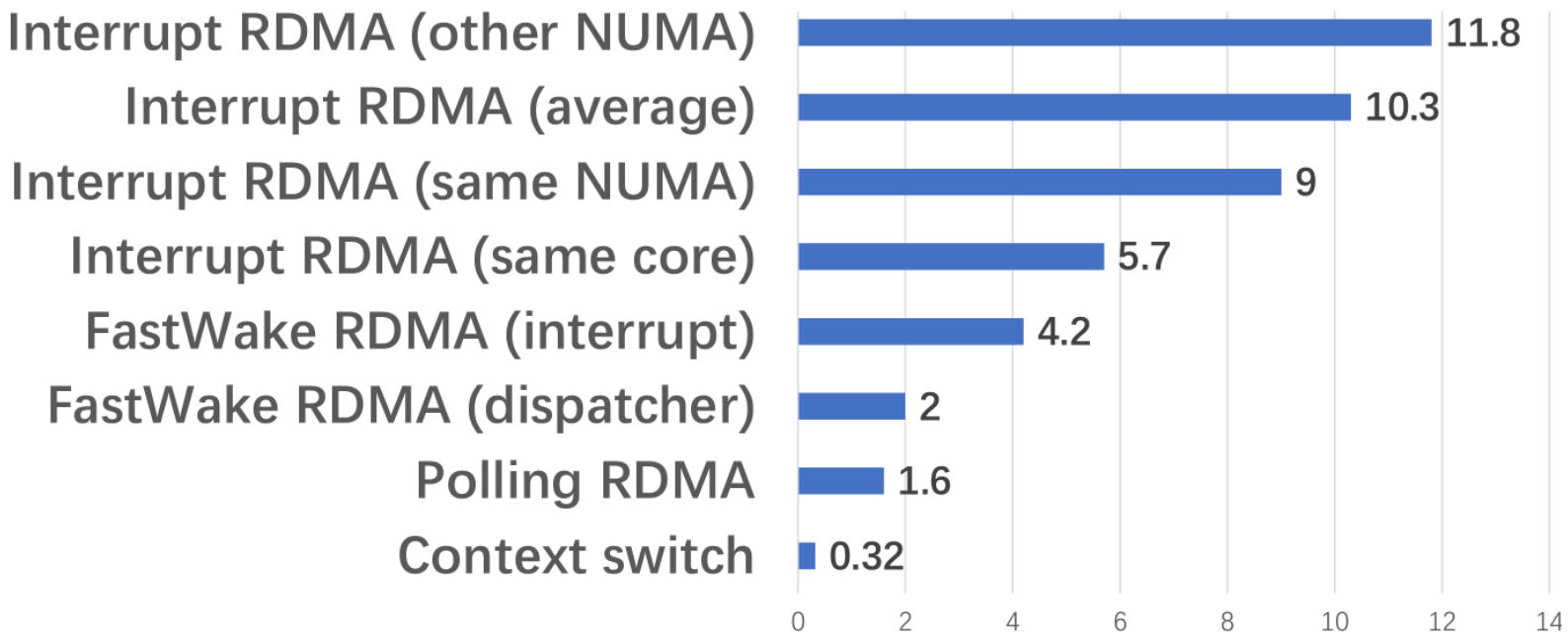
1. **Remove tasklet** and handle interrupts in top half.
   - Top-half cannot use locks and dynamically allocated memory that may lead to sleep, so we remove them.
2. Directly put the thread to wake up at the head of the runqueue, **bypassing the kernel scheduler**.
3. Other minor optimizations (see the paper).

# FastWake system architecture




- Compatible with **existing RDMA applications.**
  - Only need an LD_PRELOAD library as a shim layer.
- Compatible with **existing OS.**
  - Update OFED kernel modules to shorten thread wake-up path and implement CQ-to-EQ remapping.
  - Add a switch_to(pid) system call for direct context switch.
- Compatible with **existing RDMA NIC hardware.**

# Evaluation – Latency (x86)

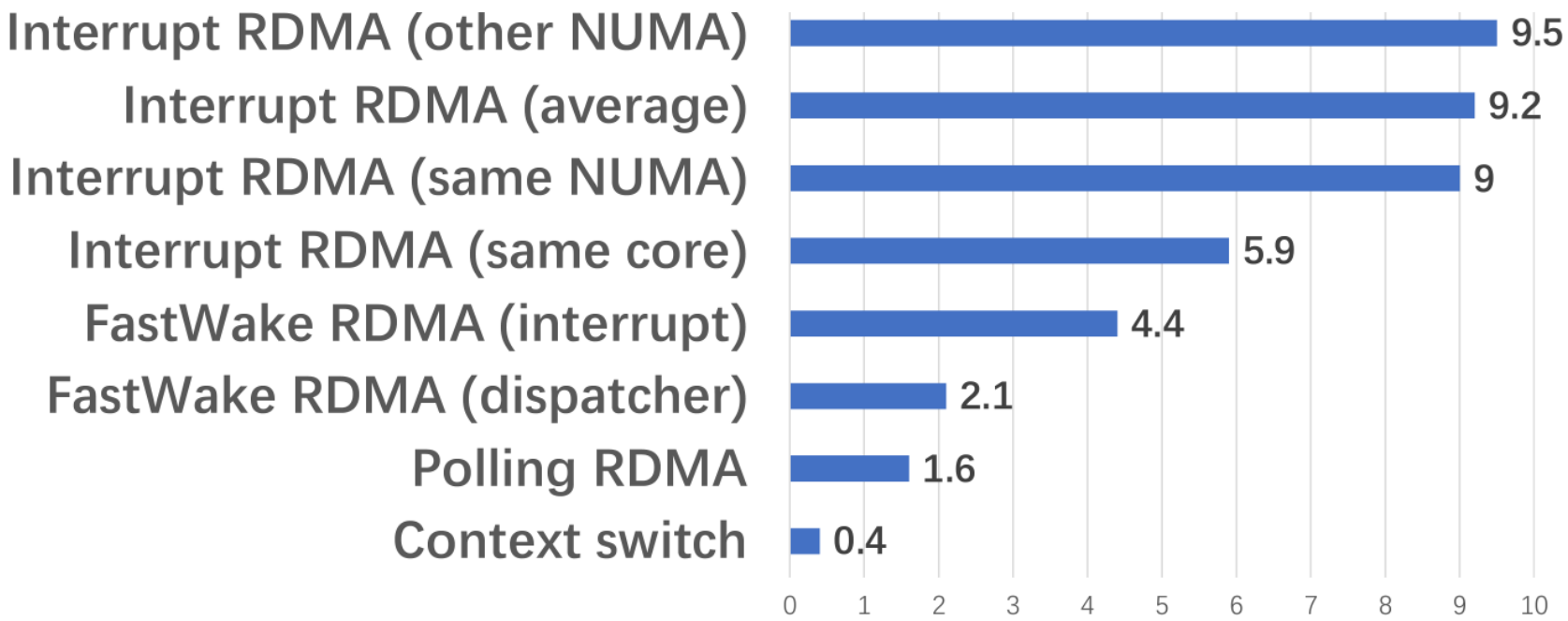



(a) X86 architecture.

**Figure 6: Latency ($\mu$s) of FastWake and traditional RDMA.**

**Dispatcher approach**: reduce RDMA latency by 65%~83% on x86 at the cost of high power utilization.
**Interrupt approach**: reduce RDMA latency by 26%~64% on x86.

# Evaluation – Latency (ARM)

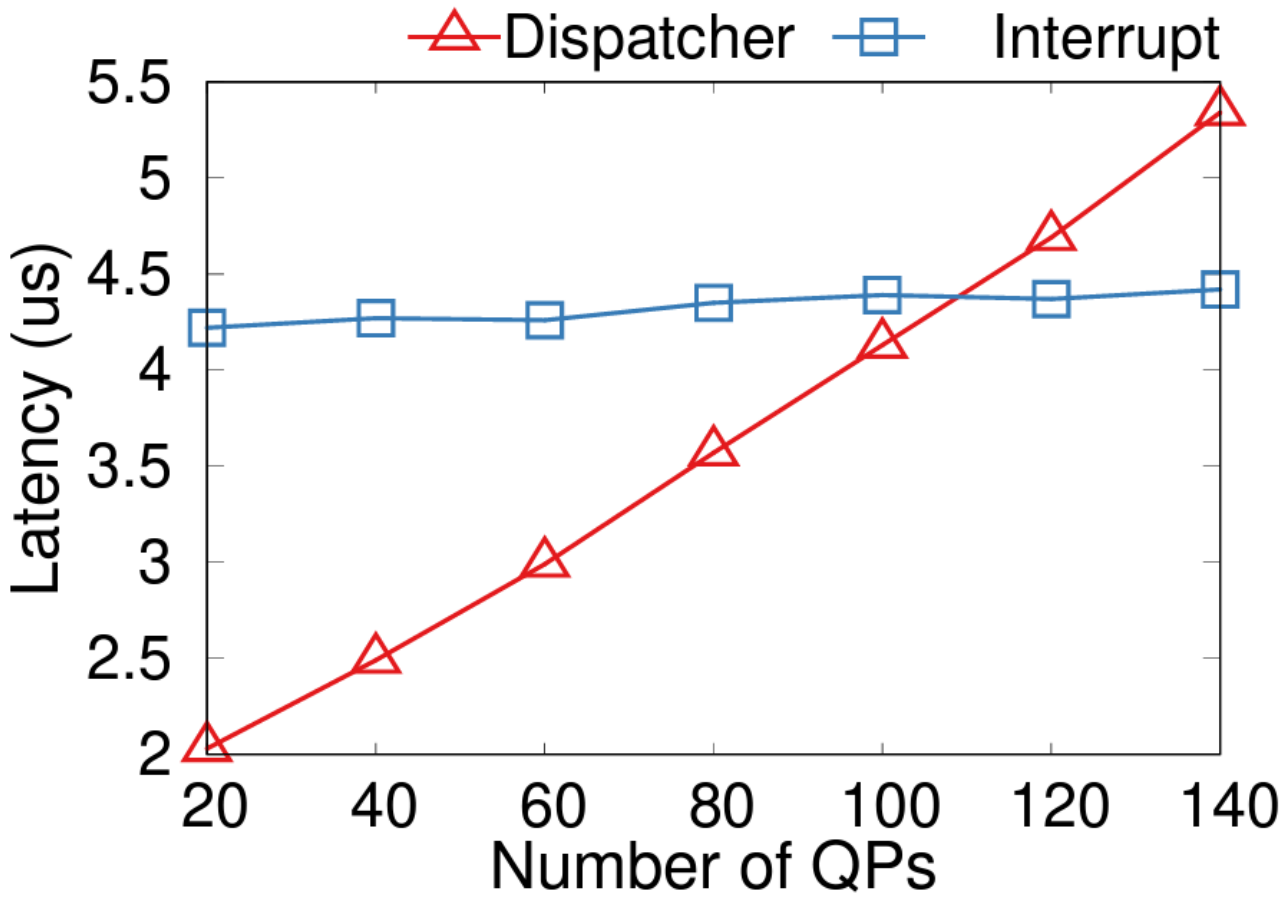



(b) ARM architecture.

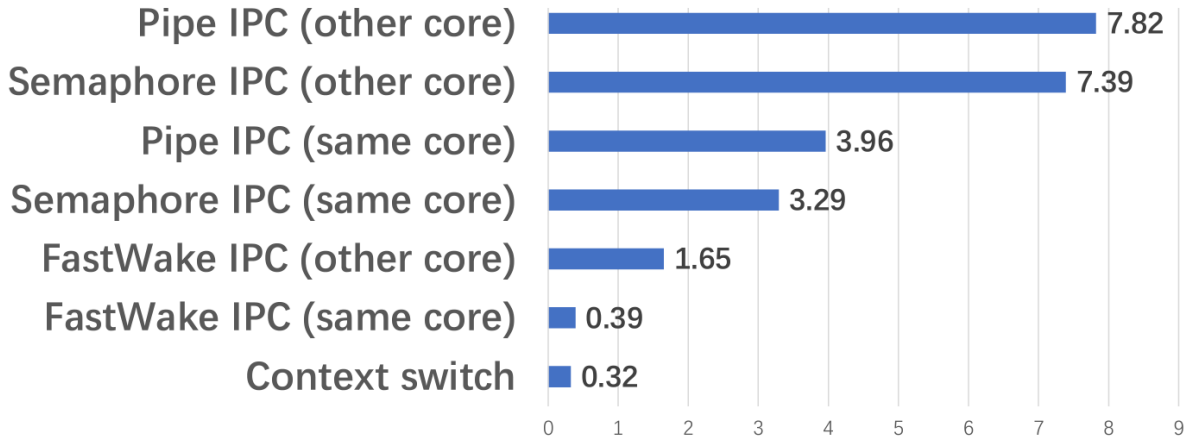
**Figure 6: Latency ($\mu$s) of FastWake and traditional RDMA.**

**Dispatcher approach**: reduce RDMA latency by 64%~78% on ARM at the cost of high power utilization.
**Interrupt approach**: reduce RDMA latency by 25%~54% on ARM.

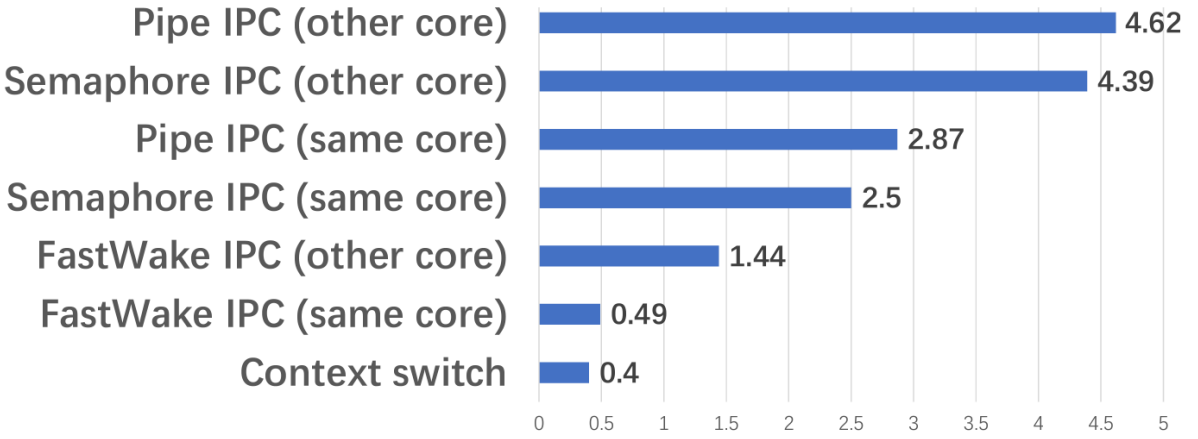
# Comparing dispatcher and interrupt approaches of FastWake



# FastWake can also reduce IPC latency




(a) X86 architecture.




(b) ARM architecture.

**Figure 9: Latency ($\mu$s) of FastWake and traditional IPC.**

IPC = Inter-Process Communication

# Conclusion

- Data center networking and storage hardware are entering an age of microsecond-scale latency.
  - Current CPU hardware and OS cannot hide this latency.
- FastWake proposes two approaches with commodity NIC, OS and applications:
  - Observation 1 – **context switch is much faster than process scheduling.**
    - Solution 1 – build a **per-core dispatcher thread** and fast context switch to fully remove the interrupt overheads.
      - Reduce RDMA latency by 65%~83% on x86 and 64%~78% on ARM.
      - Only 0.4us (20%) higher than polling mode.
  - Observation 2 – **interrupt core affinity is crucial for performance.**
    - Solution 2 – a power-saving approach to reduce interrupt latency by ensuring interrupt core affinity and shortening kernel path.
      - Reduce RDMA latency by 26%~64% on x86 and 25%~54% on ARM.
- We expect future work to evaluate FastWake on real applications.
- “The Killer Microseconds” is still an open problem.

# Thanks!

Q&A

Welcome to join/collaborate with Computer and Network Protocol Lab, Huawei.