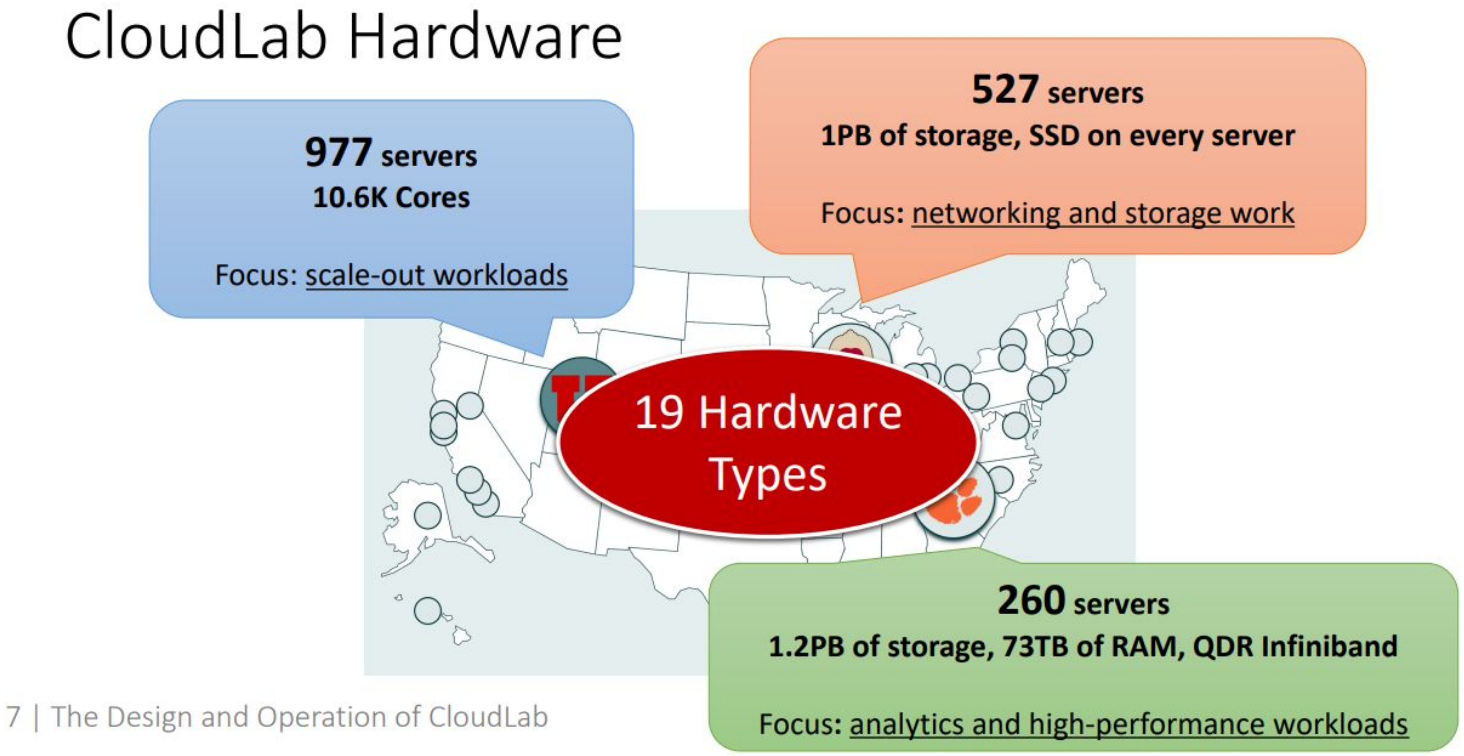
Tutorial
Cloudlab, DPDK, Shenango

MIT 6.828

Cloudlab

What Is Cloudlab?

- https://www.cloudlab.us/
- A shared cloud infrastructure for research and education in computer systems.
- The CloudLab clusters have almost 15,000 cores distributed across three sites around the United States.



The Design and Operation of CloudLab [ATC' 19]

For more information: https://docs.cloudlab.us/hardware.html

Who Use Cloudlab?

- Course instructors and students.
  - For doing lab assignments and final projects.
- System researchers.

Networking	30%
Security	16%
Storage	11%
Applications	10%
Computing	9%
Virtualization	8%
Databases	7%
Middleware	4%
Energy & Power	2%
Other	15%

* Based on 93 papers from 2017-2018

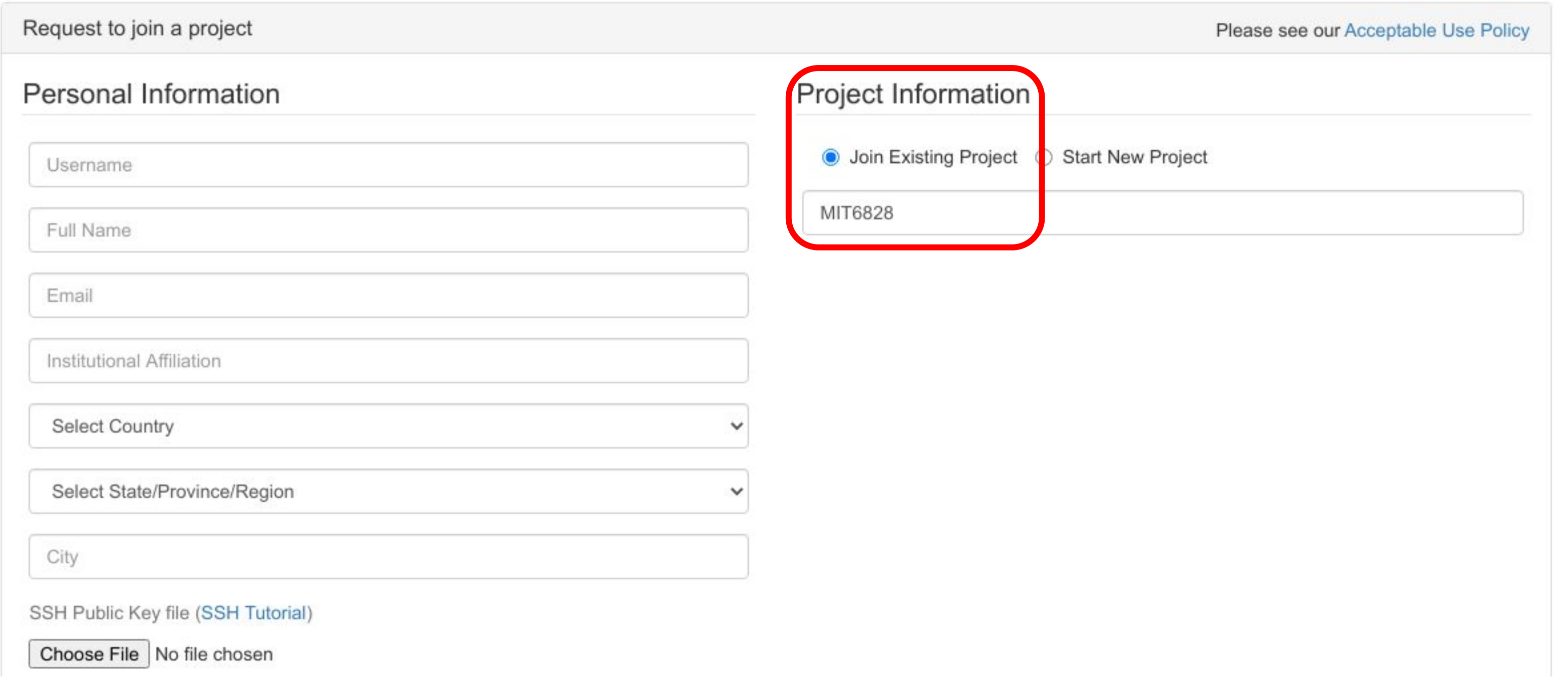
The Design and Operation of CloudLab [ATC' 19]

Why Use Cloudlab?

- Cost reasons.
  - No access to enough private servers.
  - No access to specific hardwares.
- Performance isolation.
  - System research requires accurate measurements.
  - Cloudlab provides access to **bare-metal** instances instead of VMs.
    - No interference. The instance is dedicated for your use.
- Direct hardware access.
  - For building low-latency systems.

Create Cloudlab Account

- https://www.cloudlab.us/signup.php



Launch A Cloudlab Instance

- “Start Experiment” (the most common)
  - Decide your instance type and check its availability
    https://www.cloudlab.us/resinfo.php
  - The default expiration time is 16 hours.
    - But extensions can be requested.
  - Once expired, old data are discarded.
    - Backup data. Write a script to rebuild environment automatically.
    - Or create your own disk image (snapshot).
- “Reserve Nodes”
  - For a longer machine time, e.g., one week.
  - Most reservations need cloudlab administrator’s approval.

Demo

- Launch an instance.
- Customize cloudlab profile.
- Create disk image.
- SSH example.
- Use web serial console.

Discussions

Examples:

1. How are the usage patterns similar and different between Cloudlab and a public cloud?
2. How does CloudLab maintain security?
3. Why do researchers need bare metal access to hardware? How is the hardware access provided by public clouds different?

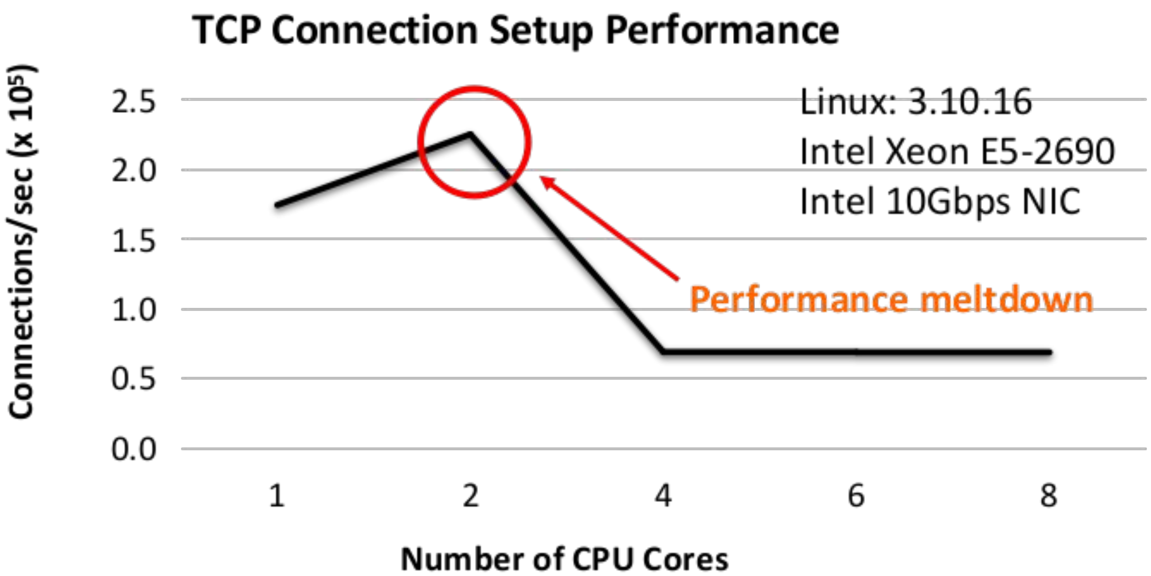
DPDK

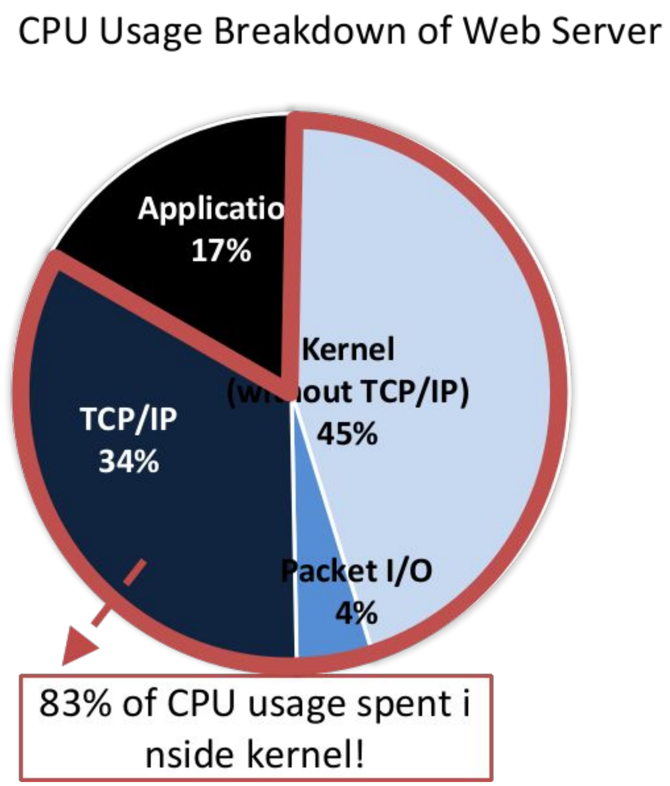
Acknowledgement: some contents are taken from DPDK's official programming guide, mTCP [NSDI' 14], and Ariel Waizel's DPDK slides.

Background

- Datacenter network is fast.
    - 100 GbE available, 200 GbE soon
- Datacenter applications need small packets.
    - Memcached used by web services. Typical KV size ~ 20 bytes.
- Requires high packet processing rate to saturate NIC.
    - 148.8 Mpps for 64 B packets in a 100 GbE link.
    - 3 GHz / 148.8 Mpps = 20 cycles per packet
    - Needs very low processing overhead, very good scalability!

Kernel Has High Overheads





Performance Bottlenecks

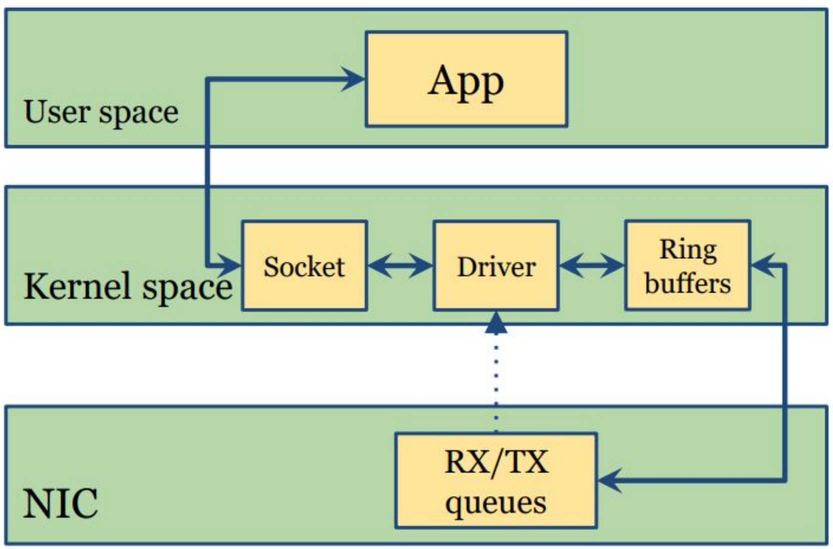
- Per packet syscall overhead.
  - Handle RX packets using interrupts.
  - BSD socket APIs.
- Shared resources.
  - Use locks on shared listening queues and file descriptor space.
  - The API is inherently unscalable pointed out by the scalable commutativity rule [SOSP' 13].
- Poor locality.
  - Interrupt handling core != application core.

DPDK Design

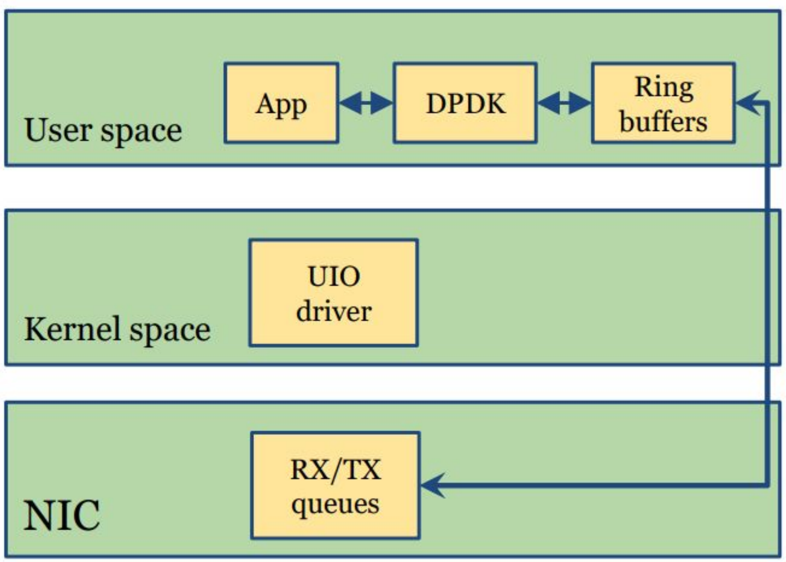
- Direct NIC queue access at user level.
  - No syscall overheads.
- Supports polling.
  - No interrupt handling overheads.
- "Share nothing" design.
  - Locks are not necessary, good for performance.
  - But harms load balancing, wastes cores.
- Supports the run-to-completion model.
  - Interleave protocol processing and application execution.
  - Good data cache locality.

DPDK Design

Packet processing in Linux



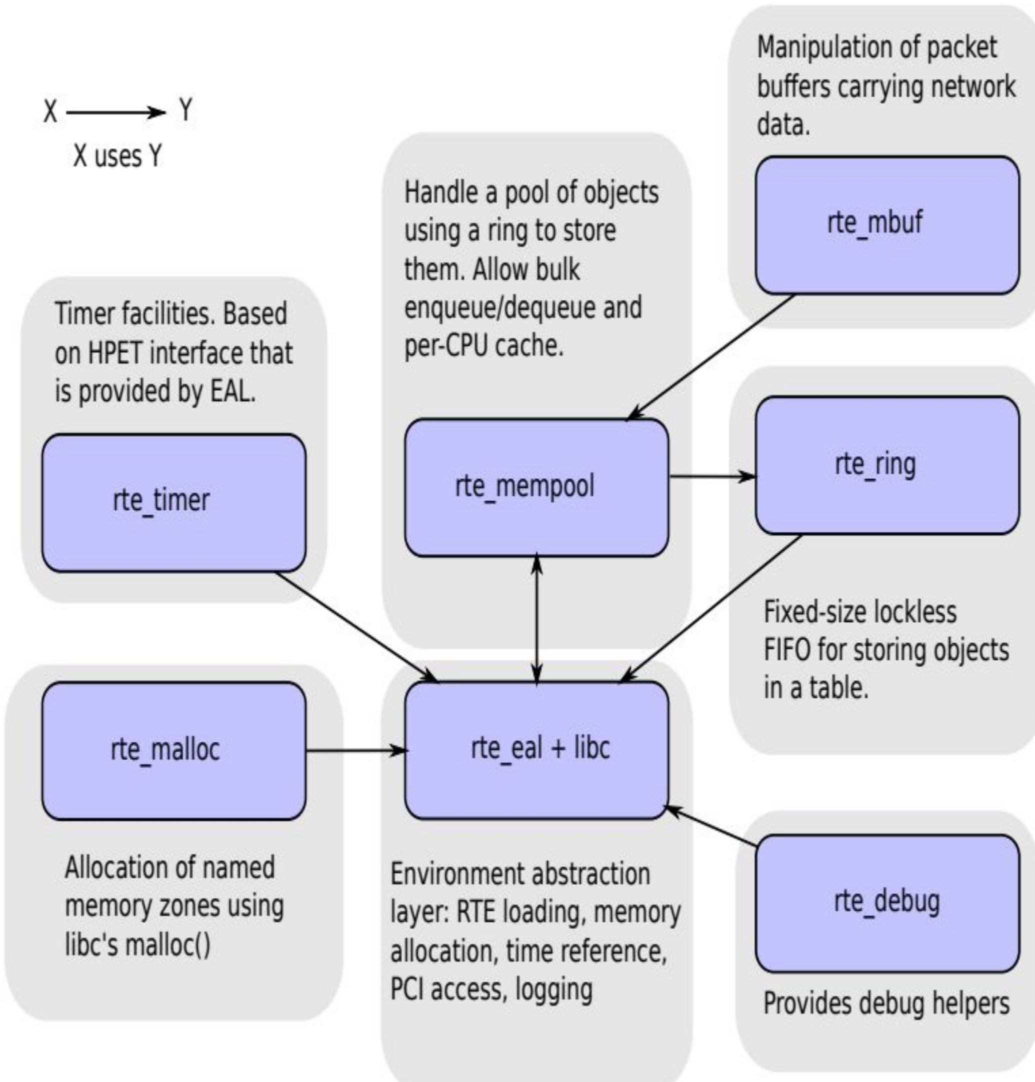
Packet processing in DPDK



DPDK Design

- Lots of performance optimizations.
  - Huge page.
  - Intel DDIO.
  - Cache alignment.
  - Thread affinity.
  - Prefetching.
  - Numa-aware memory allocation.
  - ....

Core Library Components



Code Example -- L2 Forwarding

- Forward L2 packets received from port 0 to port 1.

```
int main(int argc, char *argv[]) {
	struct rte_mempool *mbuf_pool;
	unsigned nb_ports;

	rte_eal_init(argc, argv);

	nb_ports = rte_eth_dev_count_avail();
	if (nb_ports != 2)
		rte_exit(EXIT_FAILURE, "Error: number of ports must be 2.");

	mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", NUM_MBUFS * nb_ports,
		MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());

	RTE_ETH_FOREACH_DEV(portid) {
		port_init(portid, mbuf_pool);
	}

	lcore_main();
}
```

Initialize Environment Abstraction Layer.

Allocate mbuf pool for storing RX packets.

Code Example -- L2 Forwarding

```
static inline int
port_init(uint16_t port, struct rte_mempool *mbuf_pool)
{
	struct rte_eth_conf port_conf = port_conf_default;
	const uint16_t rx_rings = 1, tx_rings = 1;
	uint16_t q;

	rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);

	for (q = 0; q < rx_rings; q++) {
		rte_eth_rx_queue_setup(port, q, RX_RING_SIZE,
				rte_eth_dev_socket_id(port), NULL, mbuf_pool);
	}

	for (q = 0; q < tx_rings; q++) {
		retval = rte_eth_tx_queue_setup(port, q, TX_RING_SIZE,
					rte_eth_dev_socket_id(port), NULL);
	}

	rte_eth_dev_start(port);

	return 0;
}
```

Set the number of RX and TX queues.

Setup the RX queue and bind it with allocated mbuf pool.

Setup the TX queue. Optionally, you can have a TX buffer (not the case here).

Start the Ethernet device.

Code Example -- L2 Forwarding

```
static __rte_noreturn void lcore_main(void) {
        for (;;) {
                RTE_ETH_FOREACH_DEV(port) {

                        struct rte_mbuf *bufs[BURST_SIZE];
                        const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                                        bufs, BURST_SIZE);

                        if (unlikely(nb_rx == 0))
                                continue;

                        const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                                        bufs, nb_rx);

                        if (unlikely(nb_tx < nb_rx)) {
                                uint16_t buf;
                                for (buf = nb_tx; buf < nb_rx; buf++)
                                        rte_pktmbuf_free(bufs[buf]);
                        }
                }
        }
}
```

A run-to-completion loop.

Receive a burst of RX packets from the RX queue of port 0.

Transmit a burst of TX packets into the TX queue of port 1.

Free any unsent packets.

Notes

- DPDK processes ethernet frames.
- If you need higher layer functionalities, you have to implement them by yourself.
  - L3 & L4 encapsulation/decapsulation.
  - Congestion control.
  - Retransmission.
  - ...

Shenango

- How to run Shenango (by HelloWorld example)
- Shenango overview (design, IOKernel, and runtime library)
  - How IOKernel, runtime library, applications interact each other
  - Packet queue, runtime thread queue, and command queue
- Topic-by-topic (its design and which code to look at)
  - Scheduler
  - Light-weight thread
  - Network stack (DPDK)
  - Storage (SPDK)
  - Synchronization (mutex, spin lock, conditional variable, rcu, etc.)
  - Timer

Shenango

What is Shenango?

- Datacenter operating system providing **low latency** and **high CPU efficiency**.
- Not a stand-alone O/S.
  - Runs on top of a standard Linux environment
  - Bypasses Linux network stack (via DPDK) and CPU scheduler
- Shenango is (mainly) composed of two components
  - **IOKernel:** re-allocates cores to application every 10 us busy-spinning in a core.
  - **Runtime library:** enables communication between applications and IOKernel and provides useful abstractions (i.e. light-weight threads)

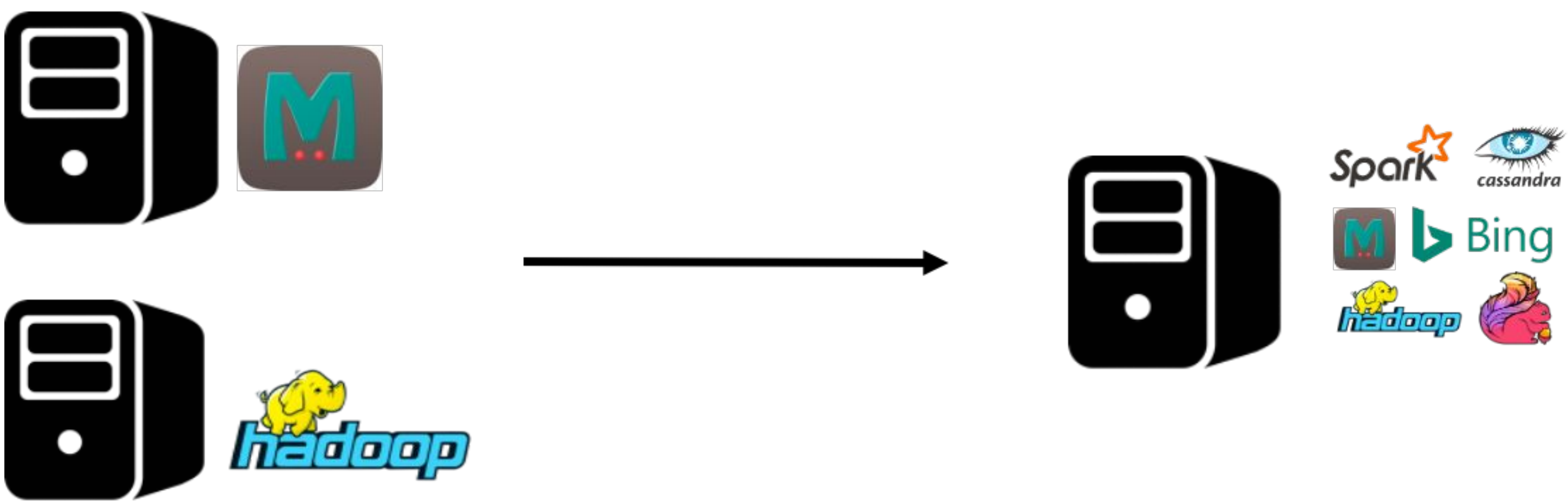
Challenge: Load variation

- Days: Diurnal ”day/night” cycles [**ADE NSDI 18**]
- Hours: Changing load composition [**ADE NSDI 18**]
- Microseconds: Packet bursts [**RZBPS SIGCOMM 15**]
- Sharding imbalances; spare capacity for failures

Problem: **Peak load** requires significantly more cores than average load

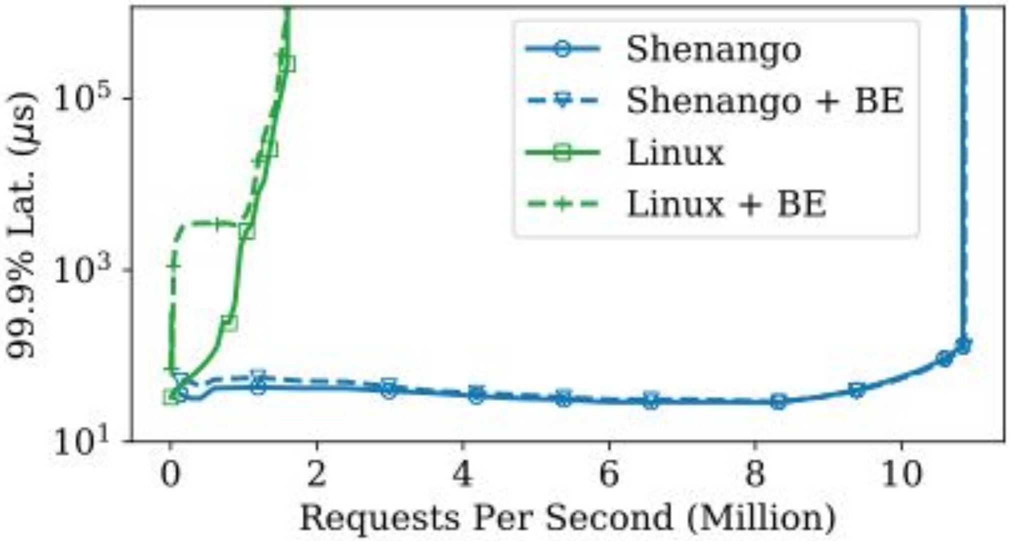
Solution: Workload consolidation

- Multiplexing between two classes of apps
  - **Latency-critical**: high-priority, given resources whenever needed
  - **Best-effort**: Low-priority, fills slack resources
- Keeps CPU load high under bursts and variability

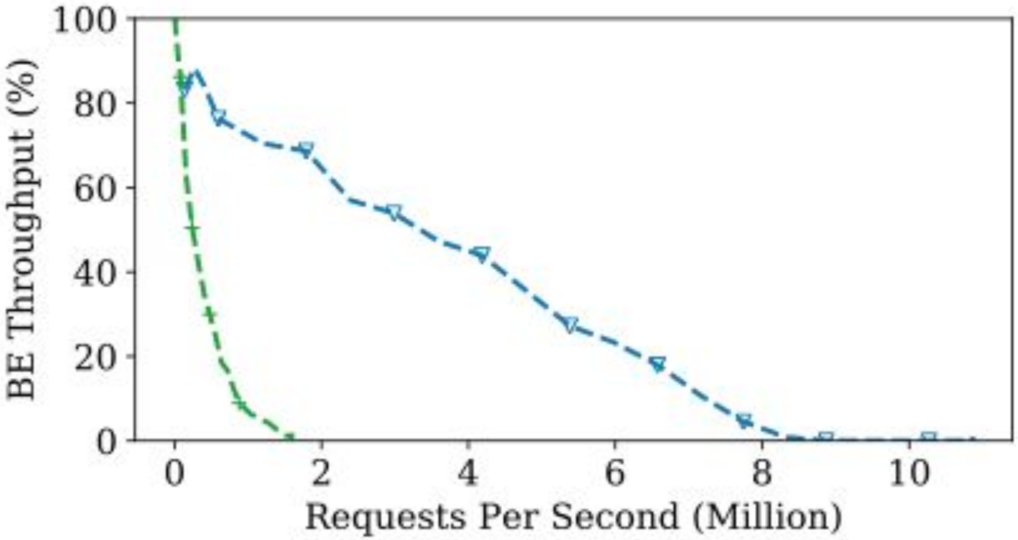


Slide from Shenango presentation in NSDI 2019

Shenango Performance Benefits

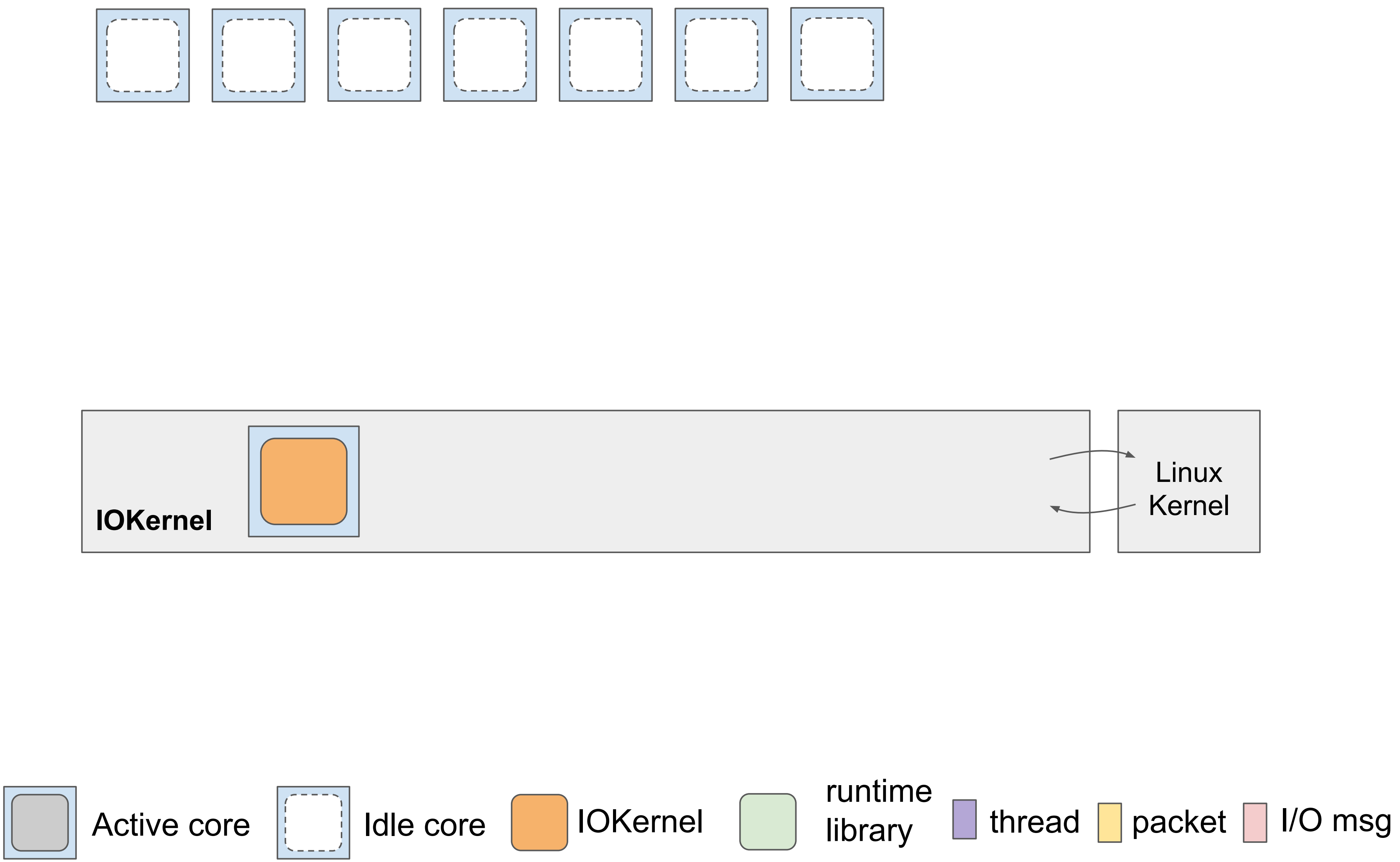


Significant reduction in latency

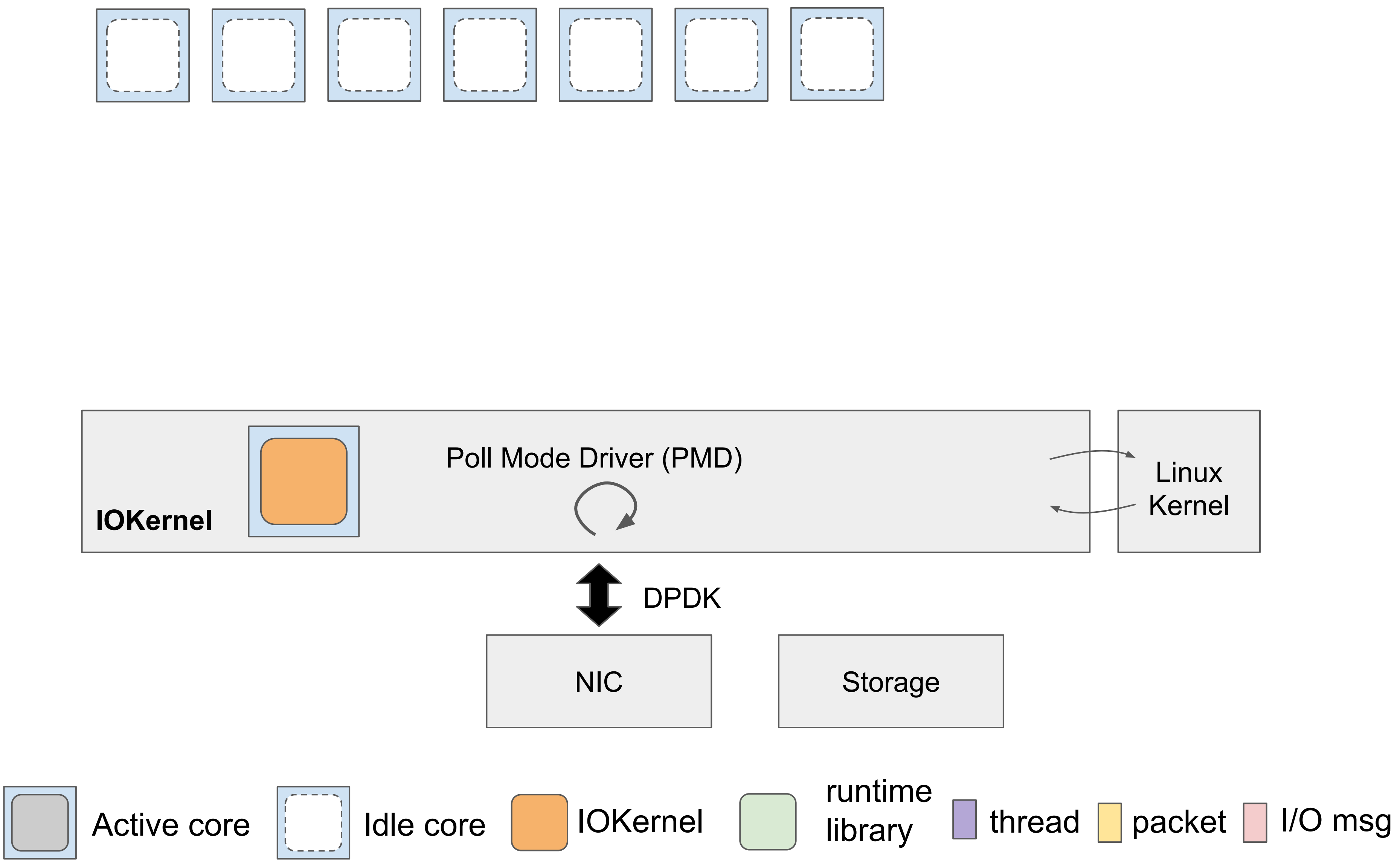


Maintains high best-effort throughput at the same time

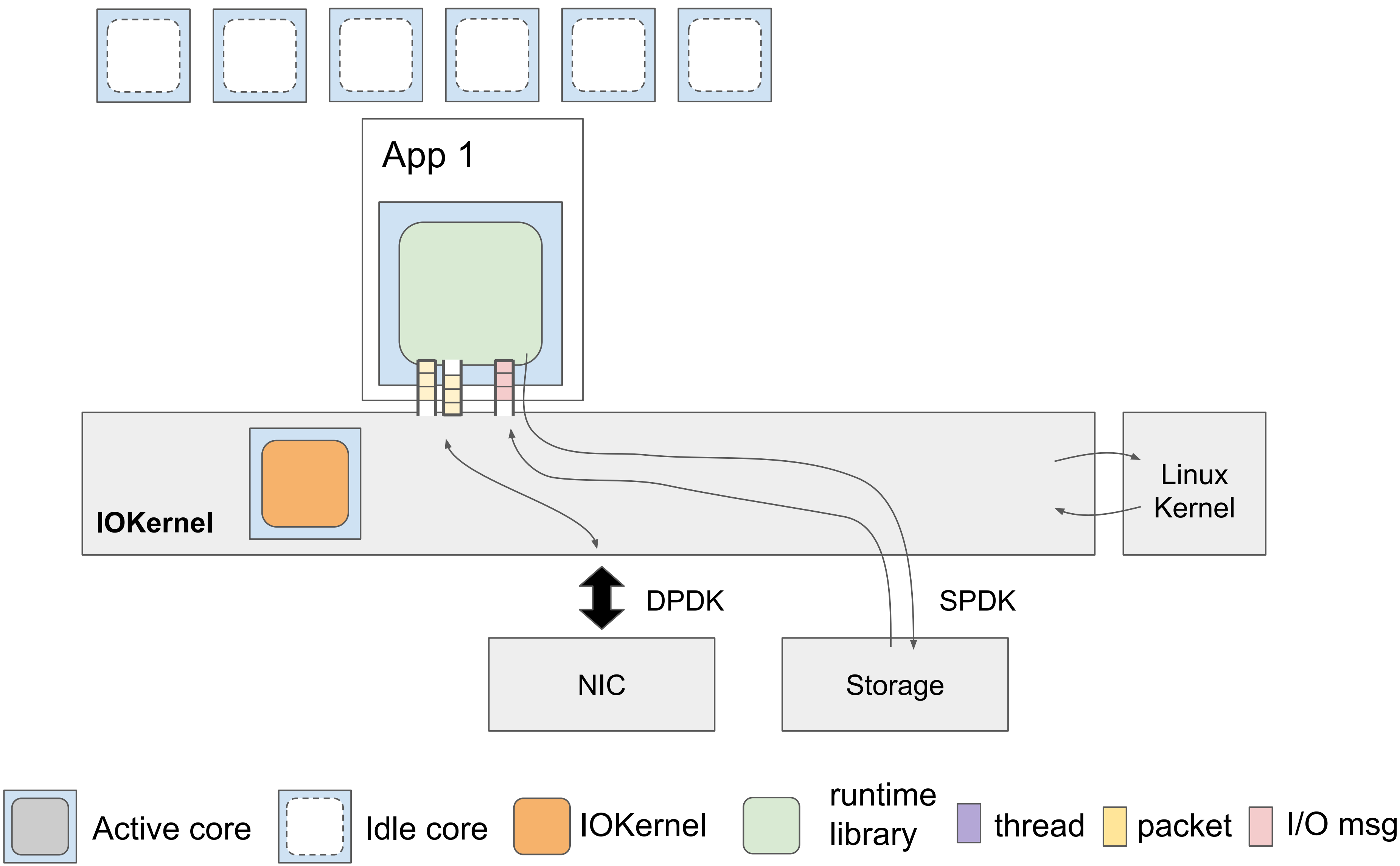
Shenango Overview



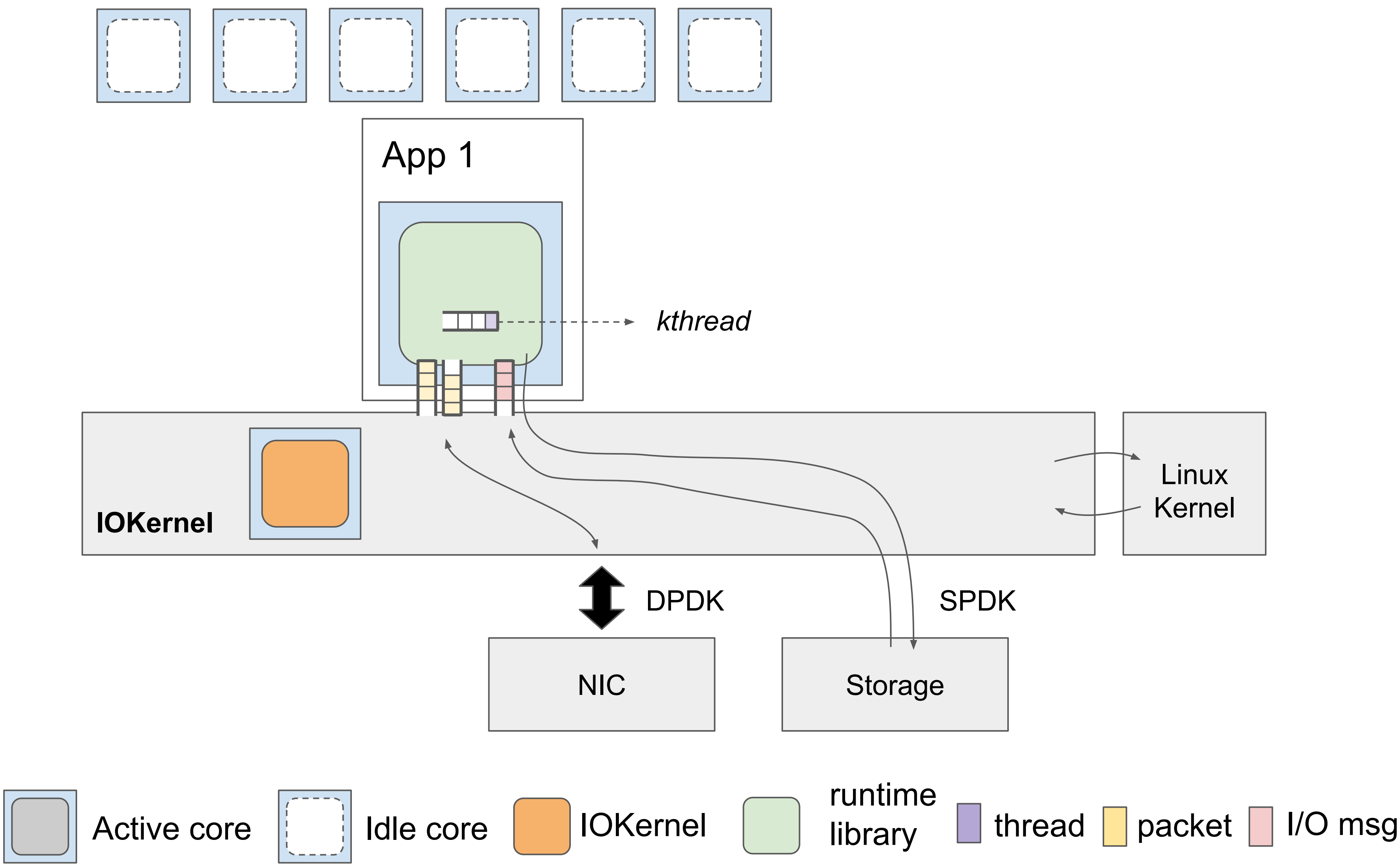
Shenango Overview



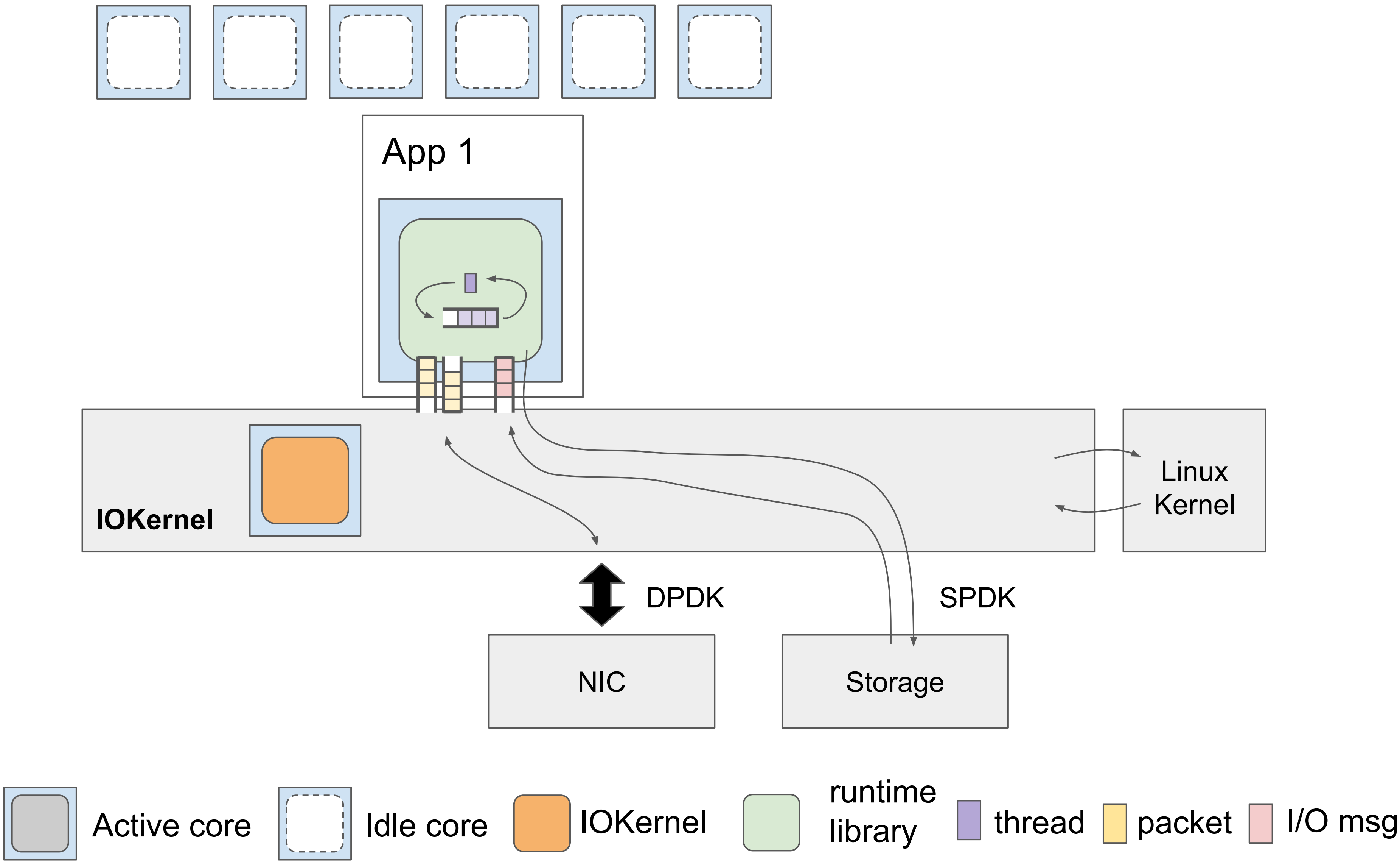
Shenango Overview



Shenango Overview

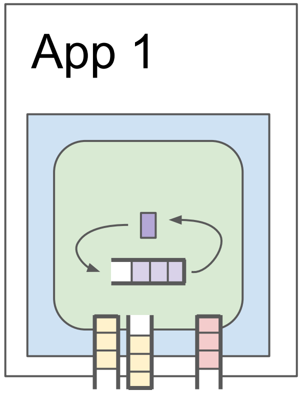


Shenango Overview



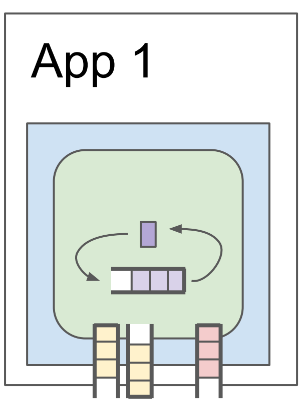
Softirq

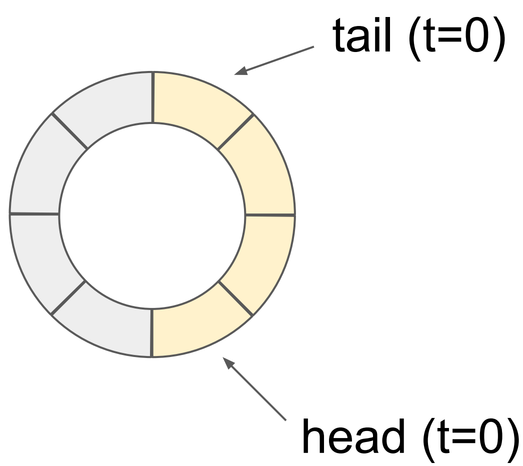
- Whenever a thread yields the core to other threads, it checks for softirq. If any, it processes them before yielding.
- Inbound packets, I/O completion message, and timer expiration is processed via softirq.



Detecting Congestion

- Queued threads or packets indicate congestion
- Any packets or threads queued since the last run (10 us ago)?
  - Grand one more core
- Ring buffers enable an efficient check
  - Head (t=n-1) > tail (t=n) implies congestion

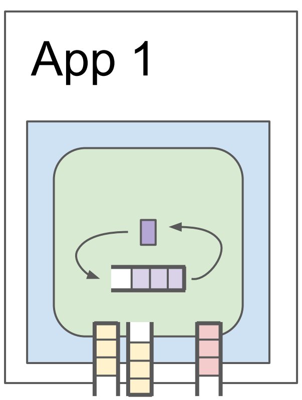


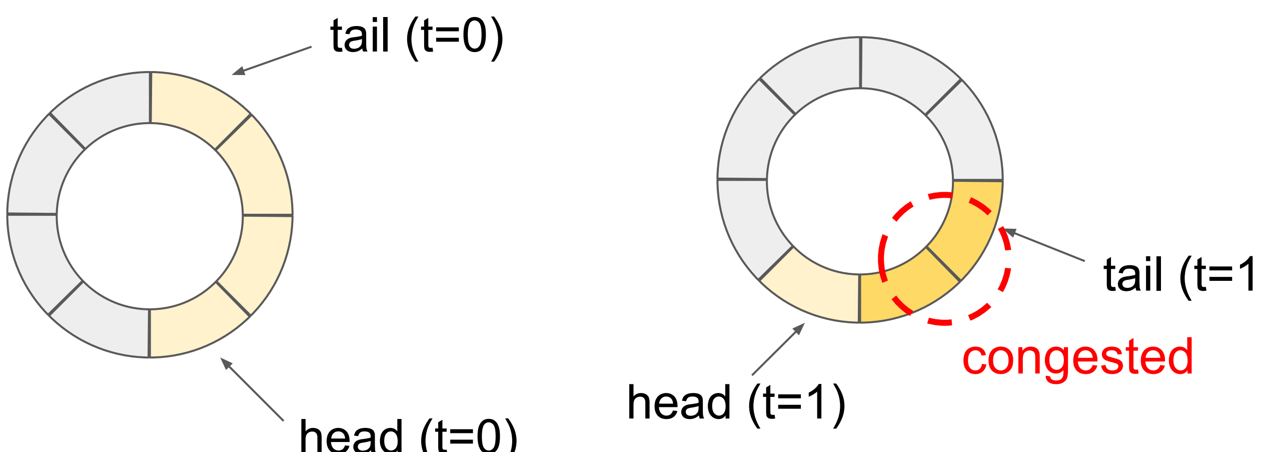


Slide from Shenango presentation in NSDI 2019

Detecting Congestion

- Queued threads or packets indicate congestion
- Any packets or threads queued since the last run (10 us ago)?
  - Grand one more core
- Ring buffers enable an efficient check
  - Head (t=n-1) > tail (t=n) implies congestion

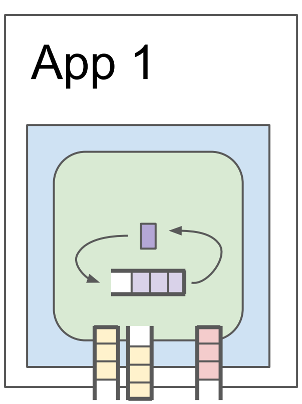


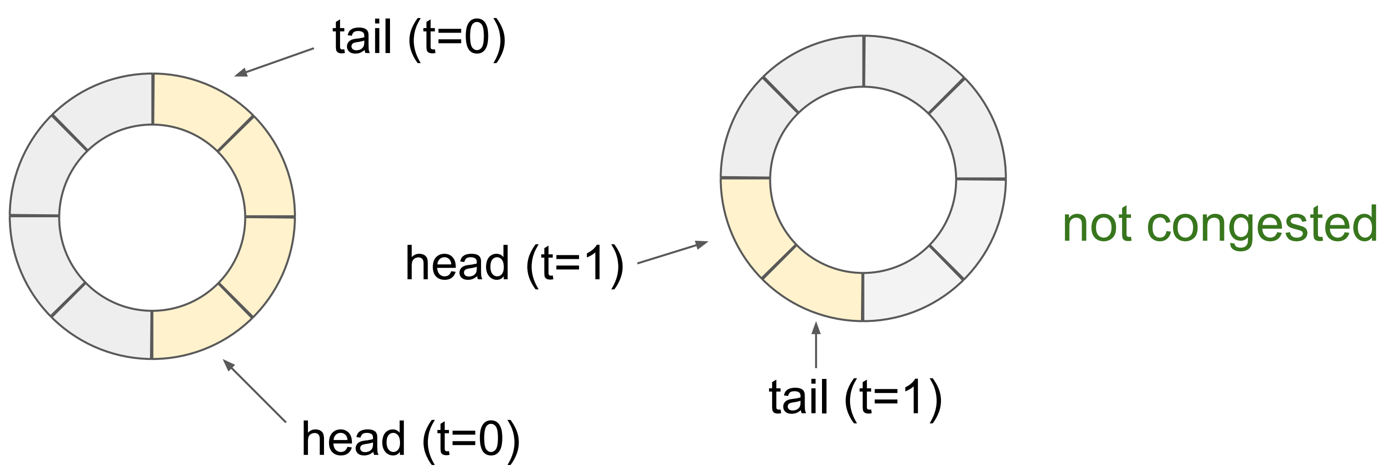


Slide from Shenango presentation in NSDI 2019

Detecting Congestion

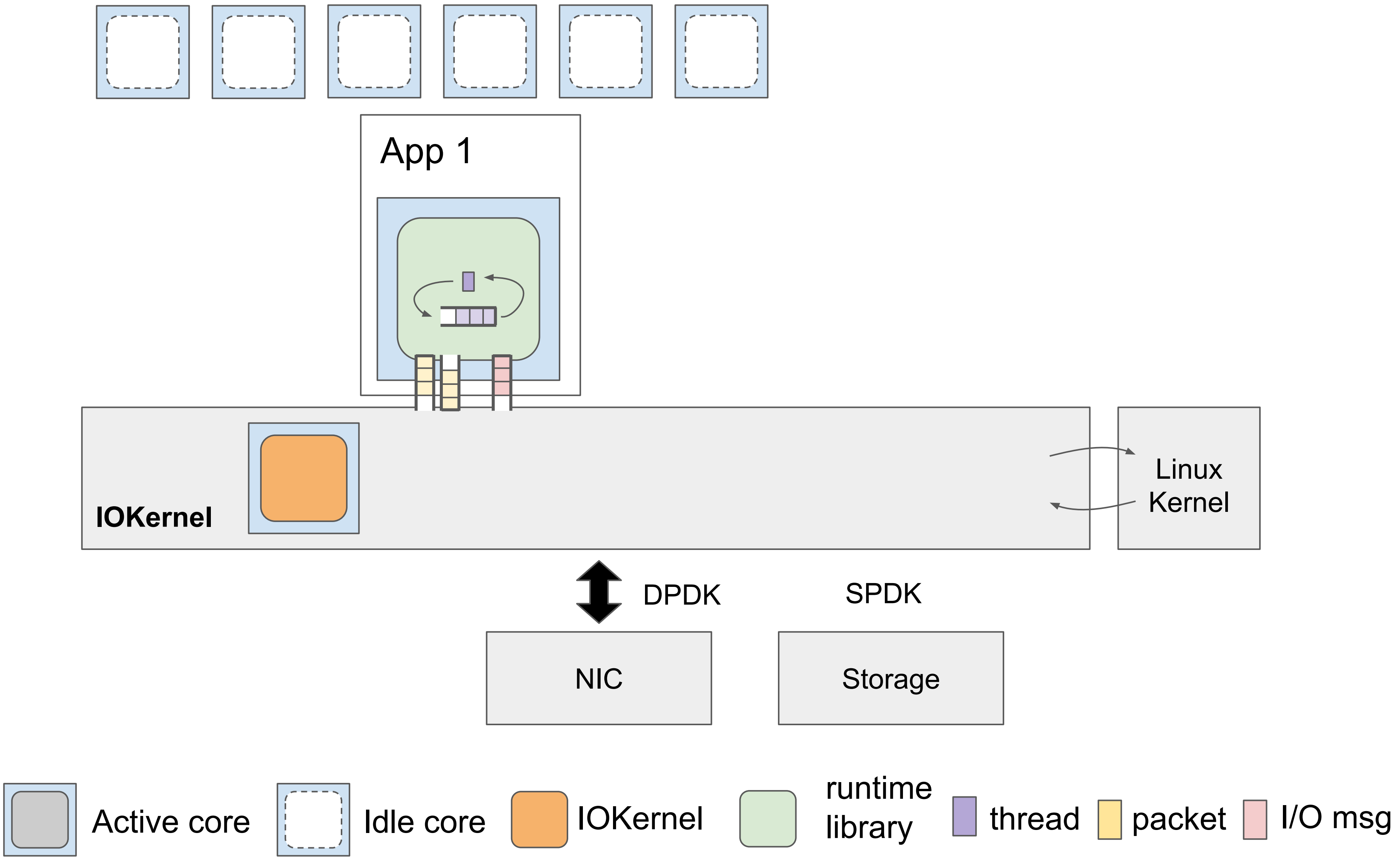
- Queued threads or packets indicate congestion
- Any packets or threads queued since the last run (10 us ago)?
  - Grand one more core
- Ring buffers enable an efficient check
  - Head (t=n-1) > tail (t=n) implies congestion



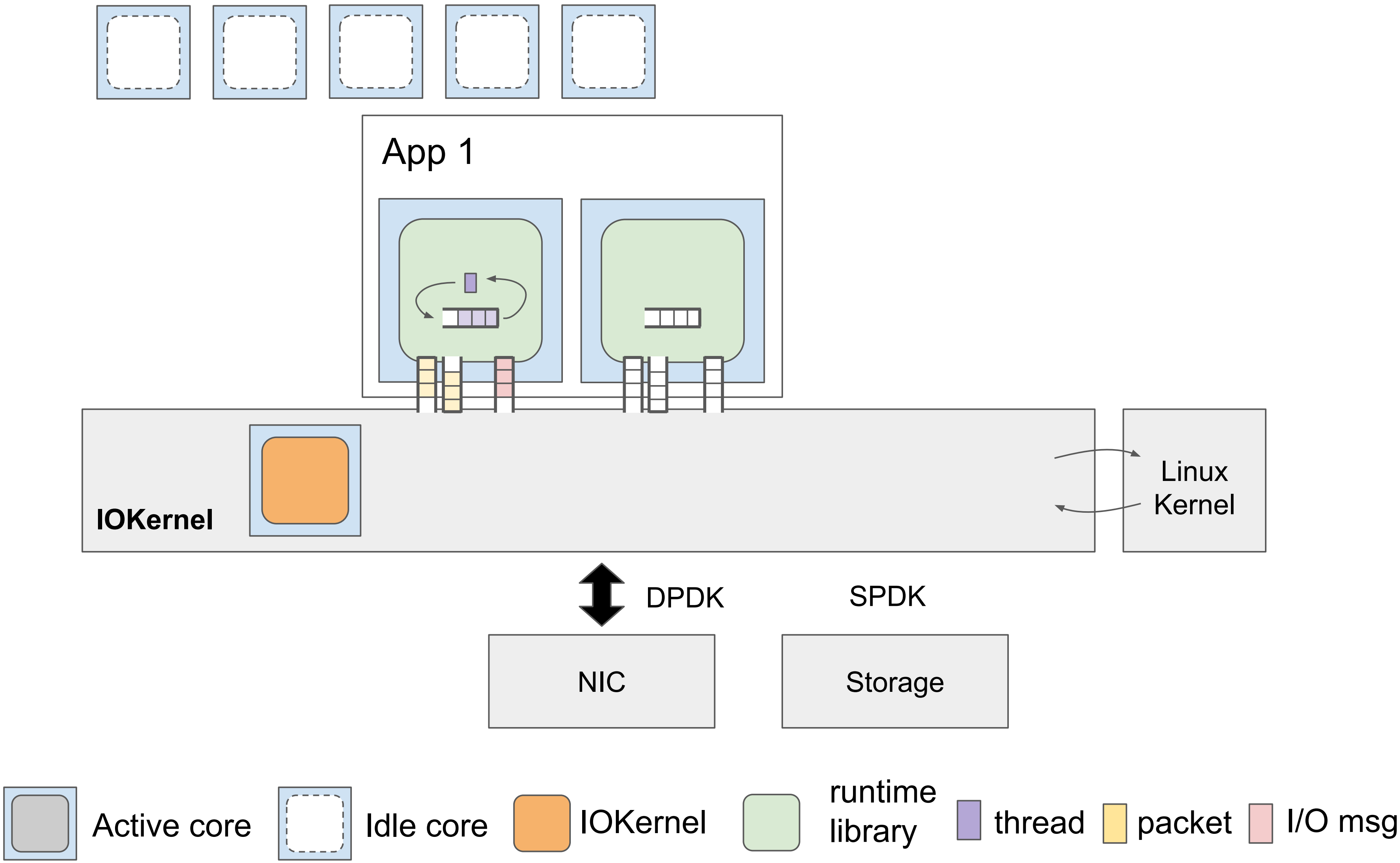


Slide from Shenango presentation in NSDI 2019

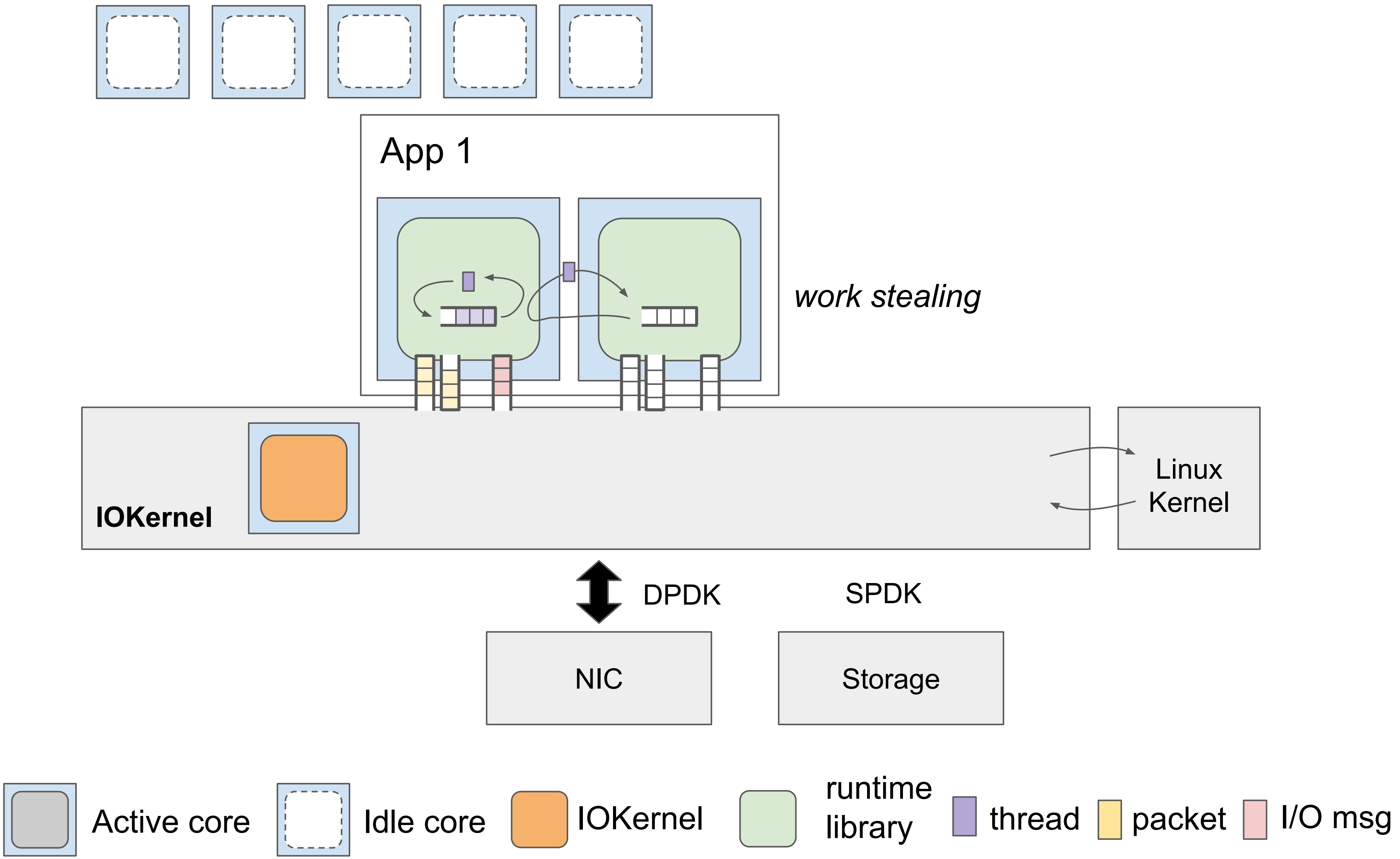
Shenango Overview



Core-allocation based on congestion



Core-allocation based on congestion



Scheduling

If you are interested in scheduling, see

- iokernel/sched.c[.h]
- iokernel/simple.c

How to run Shenango?

app1.cc

```
#include <runtime.h>
#include <iostream>

void Handler(void *arg) {
  std::cout << "Hello Shenango!" <<
std::endl;
}

int main(int argc, char *argv[]) {
  ret = runtime_init(argv[1], Handler, NULL);
  if (ret) {
    std::cerr << "failed to start runtime"
              << std::endl;
  }

  return 0;
}
```

app1.config

```
# network config
host_addr 192.168.1.100
host_netmasek 255.255.255.0
host_gateway 192.168.1.1
# runtime config
runtime_kthreads 3
runtime_spinning_kthreads 1
```

How to run Shenango?

1. Start IOKernel daemon

```
shenango$ sudo ./iokerneld
```

2. Build application
   - You should link runtime library, bindings to C++/Rust (if your application is written in C++/Rust) to compile.
3. Start application with config

```
shenango/apps/app1$ ./app1 app1.config
```

Sample applications and Makefiles are available in apps/

One more step to Shenango

- ~~Scheduling~~
- Light-weight threads
- Network stack
- Storage
- Timer
- Synchronization

Light-weight threads

app2.cc

```
#include <runtime.h>
#include <thread.h>
#include <iostream>

void Handler(void *arg) {
  std::cout << "Hello Shenango!" <<
std::endl;
  rt::Thread([] {
      std::cout << "Hello Shenango thread!"
                << std::endl;
  }
}

int main(int argc, char *argv[]) {
  ret = runtime_init(argv[1], Handler, NULL);
  if (ret) {
    std::cerr << "failed to start runtime"
              << std::endl;
  }

  return 0;
}
```

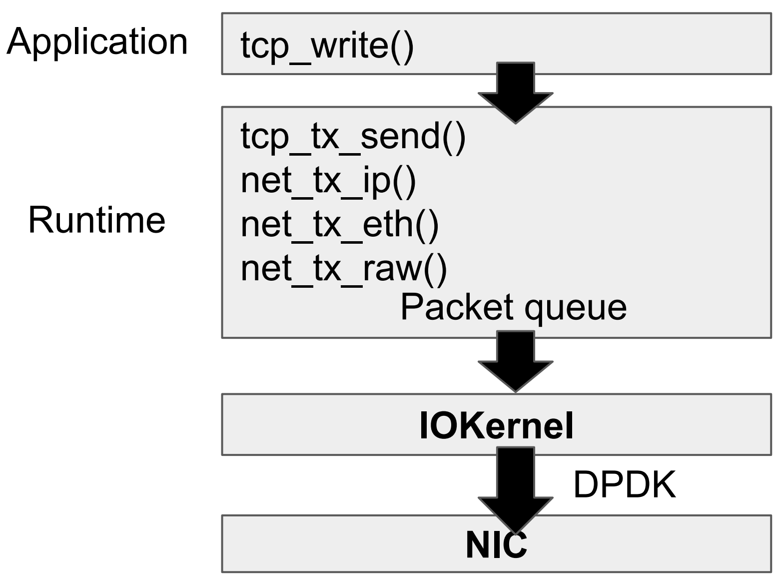
Light-weight threads

If you are interested in scheduling, see

- inc/runtime/thread.h
- base/thread.c

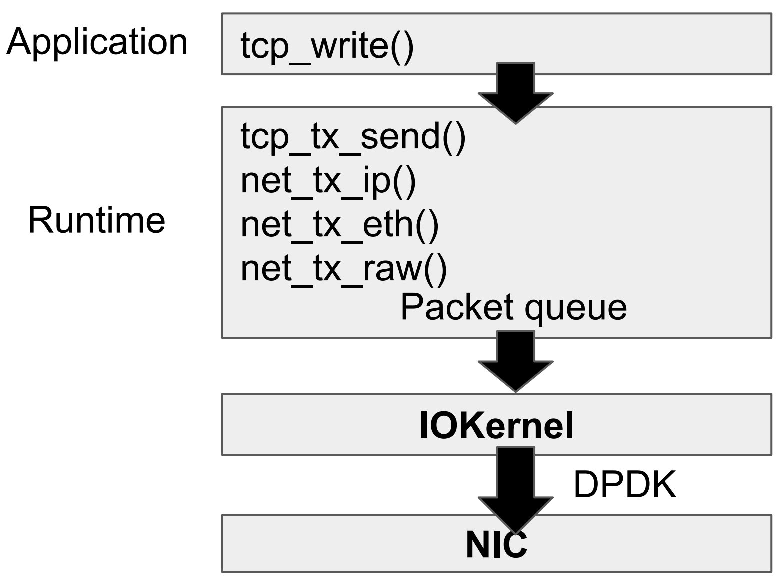
Network Stack

- Incoming packets are processed via softirq
- Whole network stack is implemented in runtime library.
- It supports Ethernet, ARP, ICMP, IP, TCP, UDP.
- If you are interested in Shenango network stack, see
    - inc/net/
    - runtime/net/

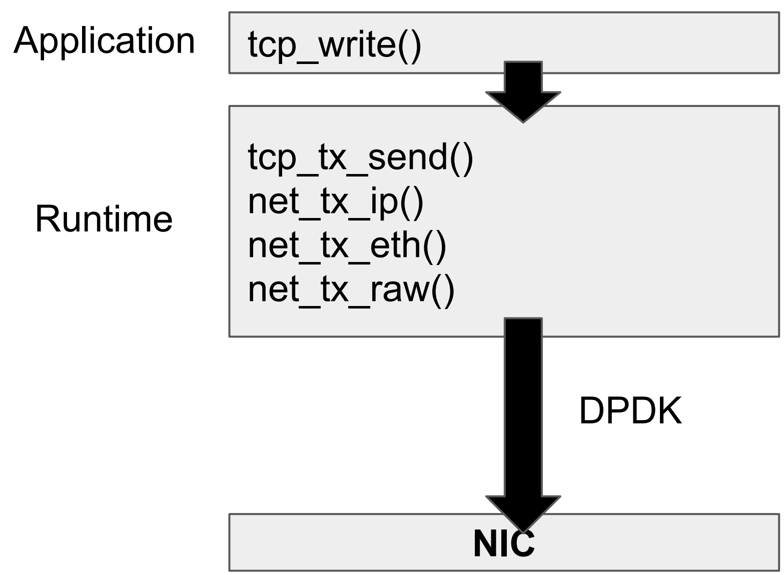


Network Stack

- Incoming packets are processed via softirq
- Whole network stack is implemented in runtime library.
- It supports Ethernet, ARP, ICMP, IP, TCP, UDP.
- If you are interested in Shenango network stack, see
  - inc/net/
  - runtime/net/

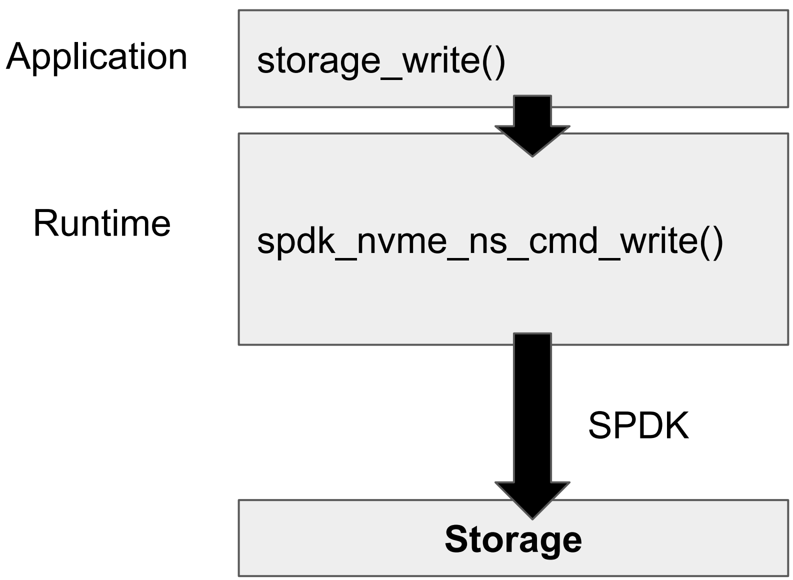


(With Direct Path)



Storage

- Inbound I/O completion messages are processed via softirq
- SPDK is a DPDK-equivalent library for storage.
- Unlike network stack, runtime library directly issue SPDK command to storage device (No I/O kernel involved).



Timer

- Timer expiration is handled by softirq and the thread with expired timer is enqueued to the thread queue
- If you are interested in scheduling, see
  - inc/runtime/timer.h
  - runtime/timer.c

Synchronization

- Shenango runtime supports
  - Mutex
  - Conditional variables
  - Spin lock
  - Barrier
  - Read-write mutex

- If you are interested in scheduling, see
  - inc/runtime/sync.h
  - runtime/sync.c

Shenango code

Check out https://github.com/shenango/shenango

New code release available in the next couple weeks