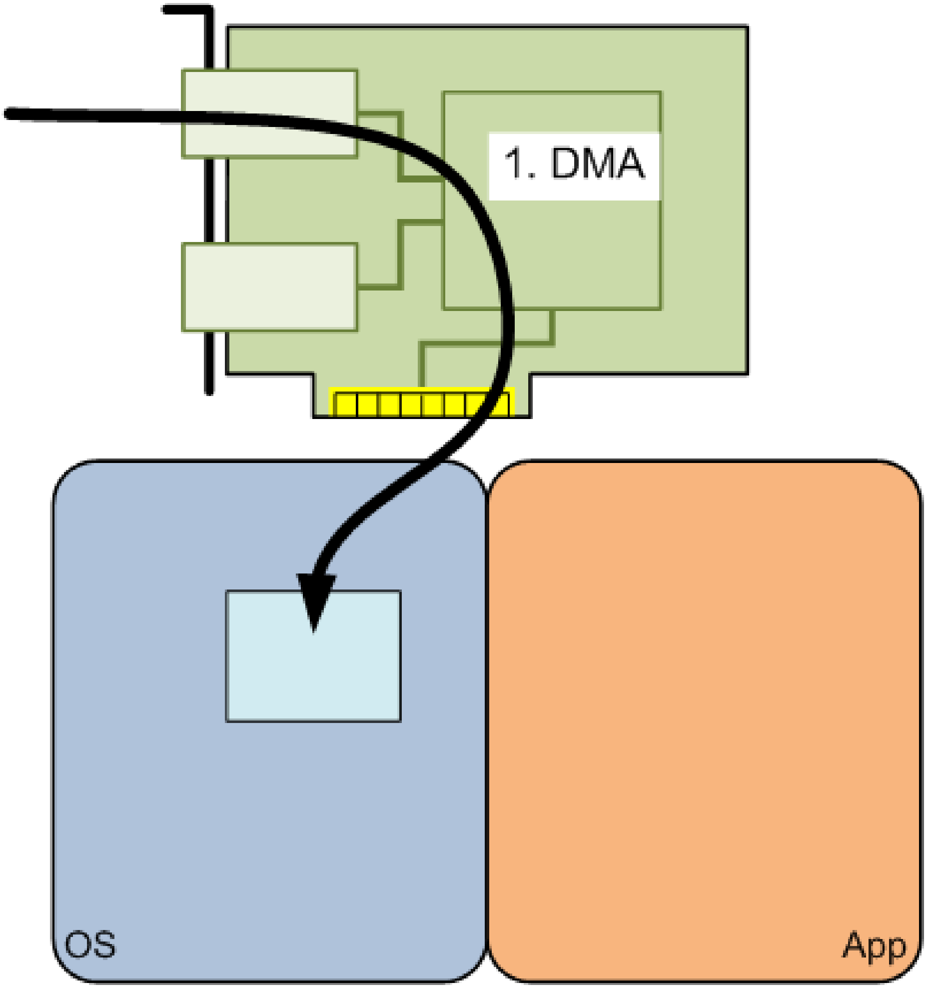
Much Faster Networking

SOLARFLARE®

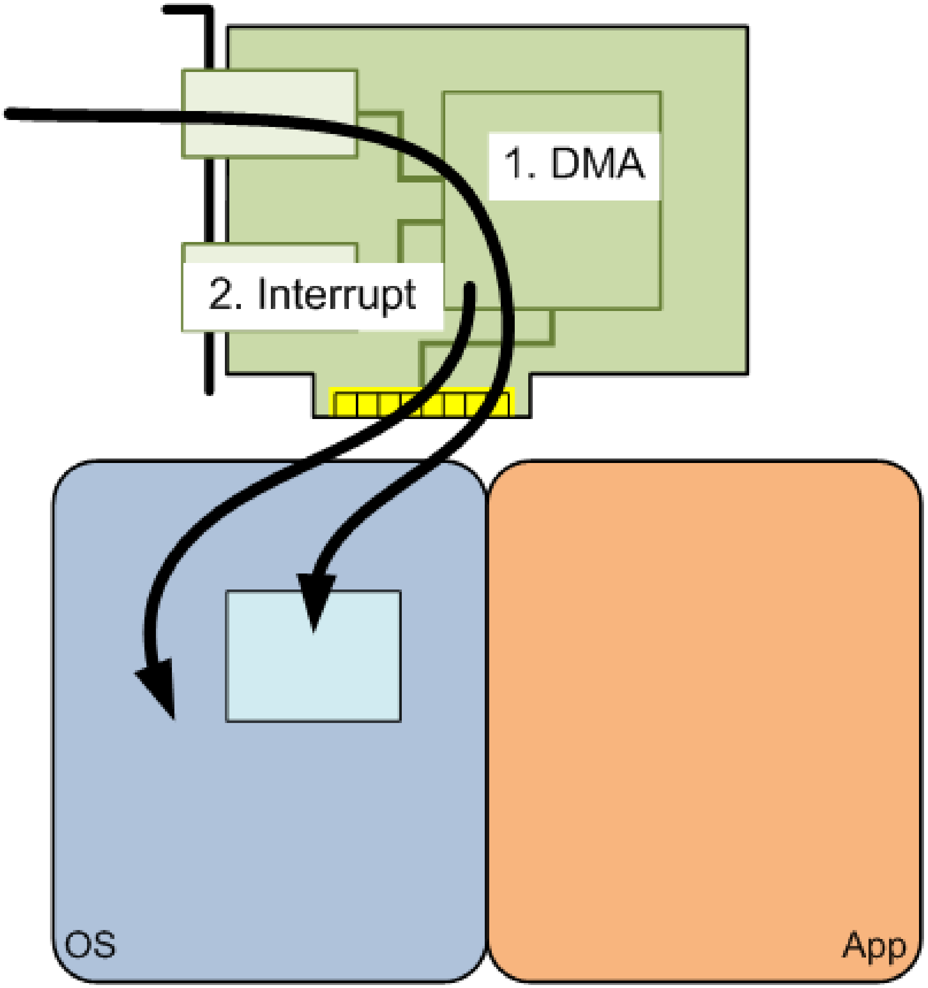
David Riddoch
driddoch@solarflare.com

What is kernel bypass?

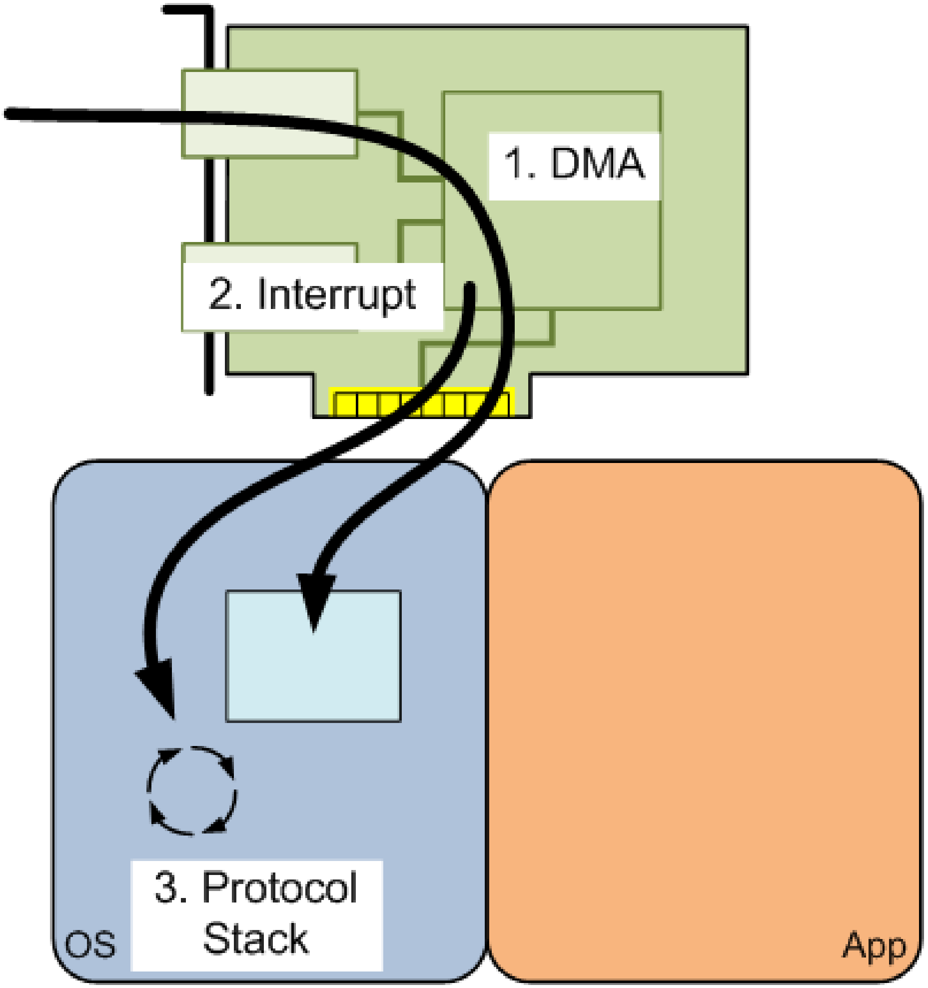
The standard receive path



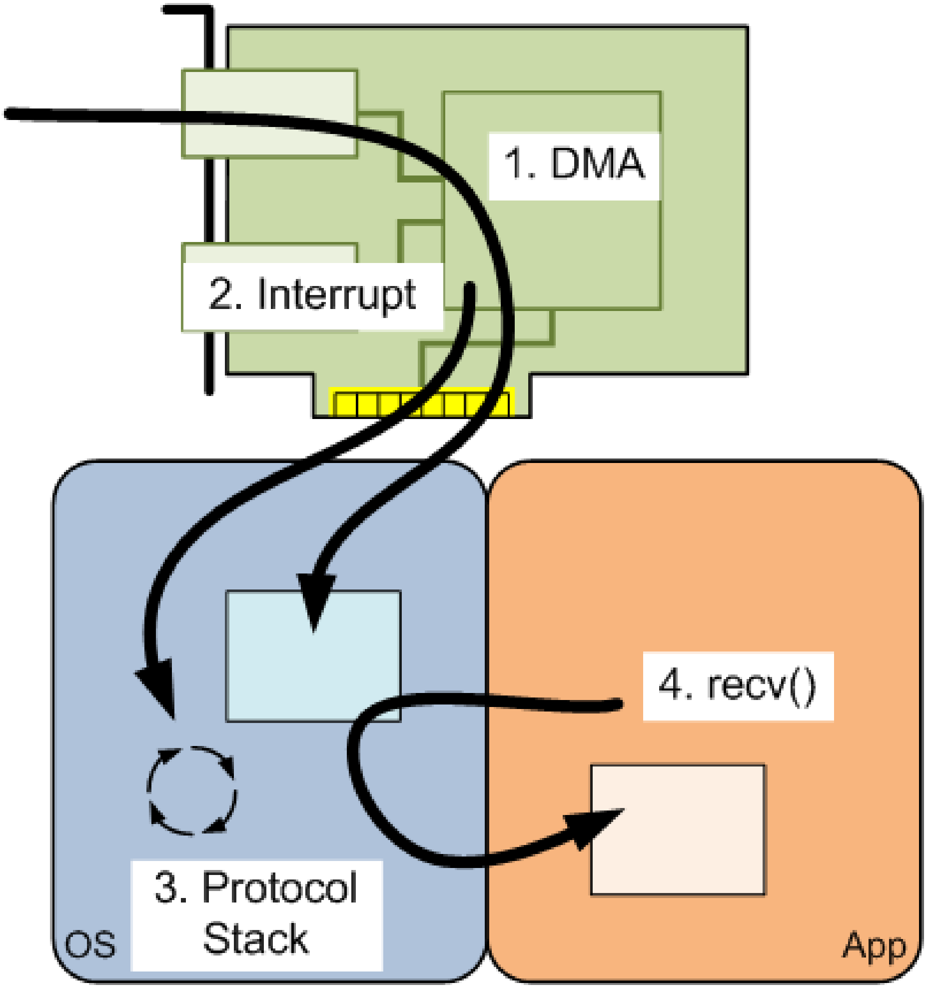
The standard receive path



The standard receive path



The standard receive path



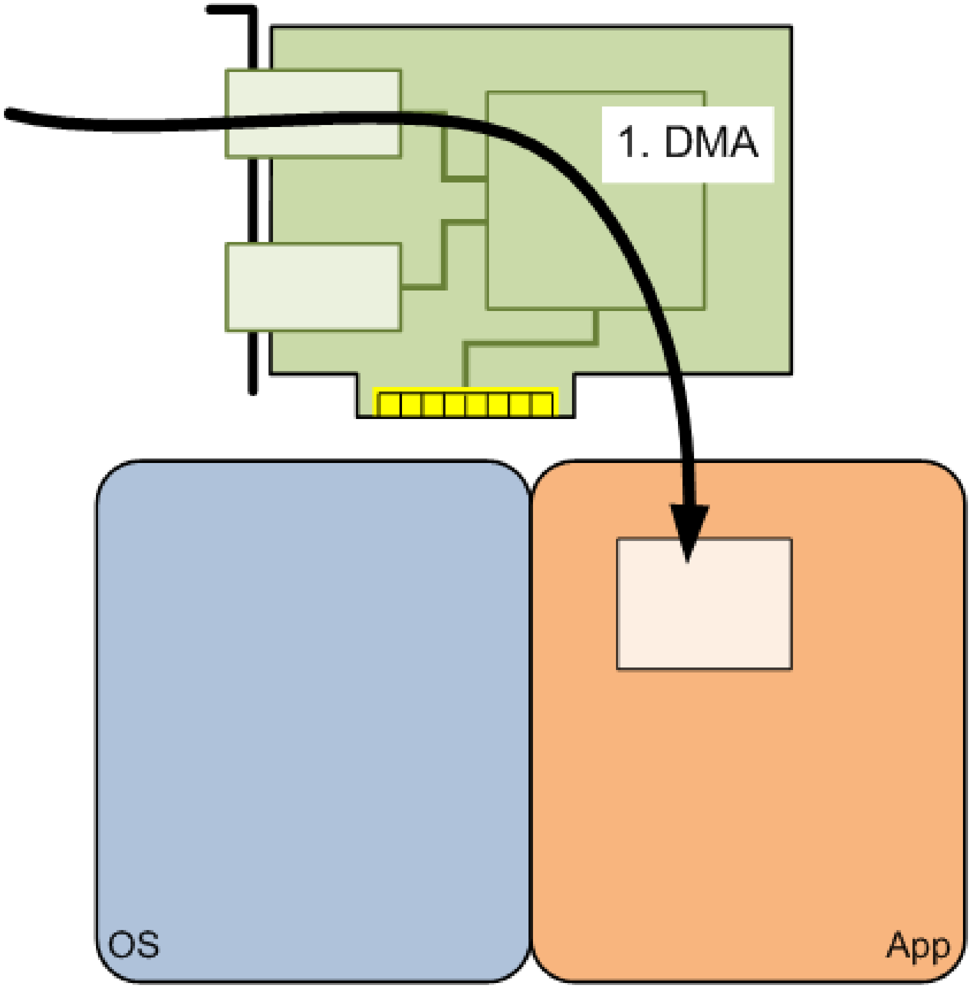
1. DMA

2. Interrupt

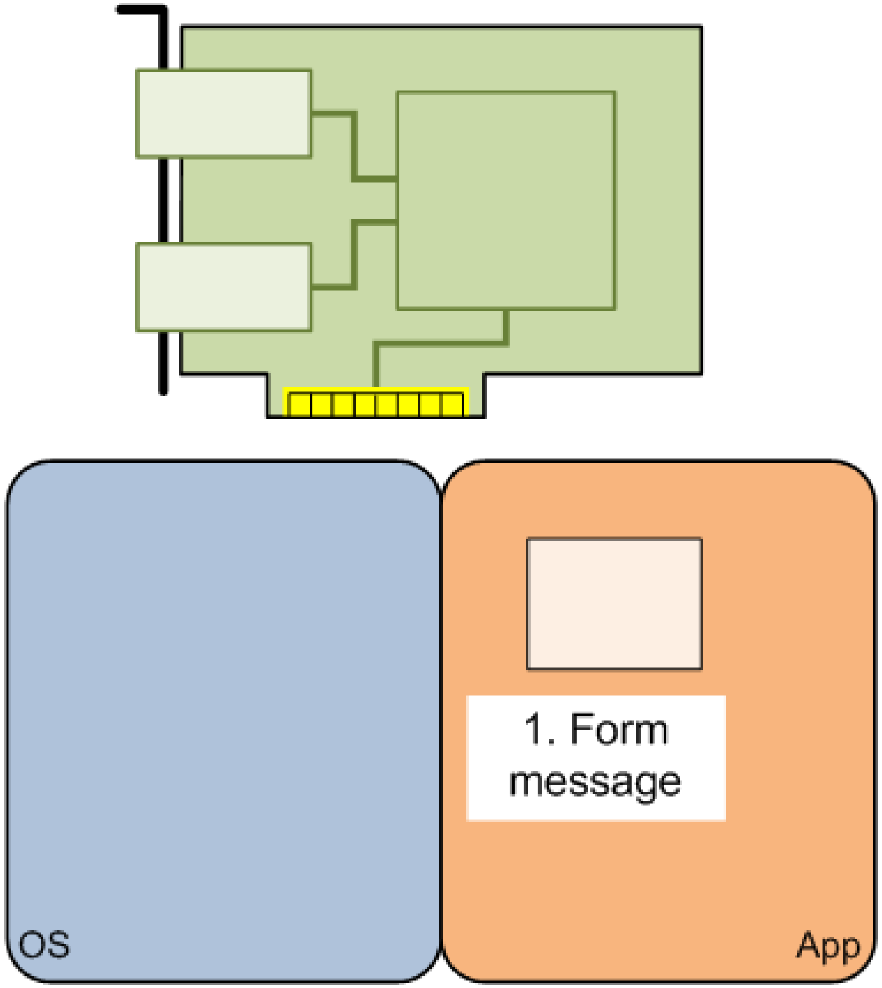
3. Protocol Stack

4. recv()

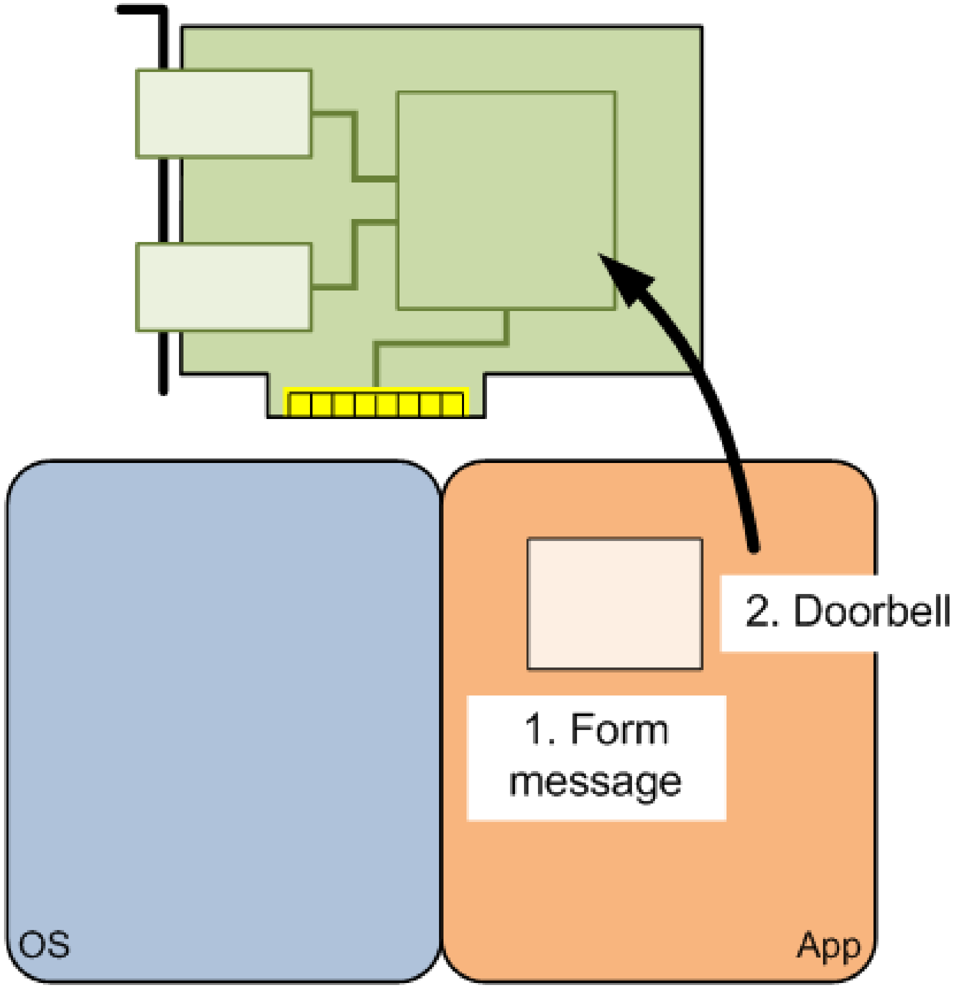
Kernel-bypass receive



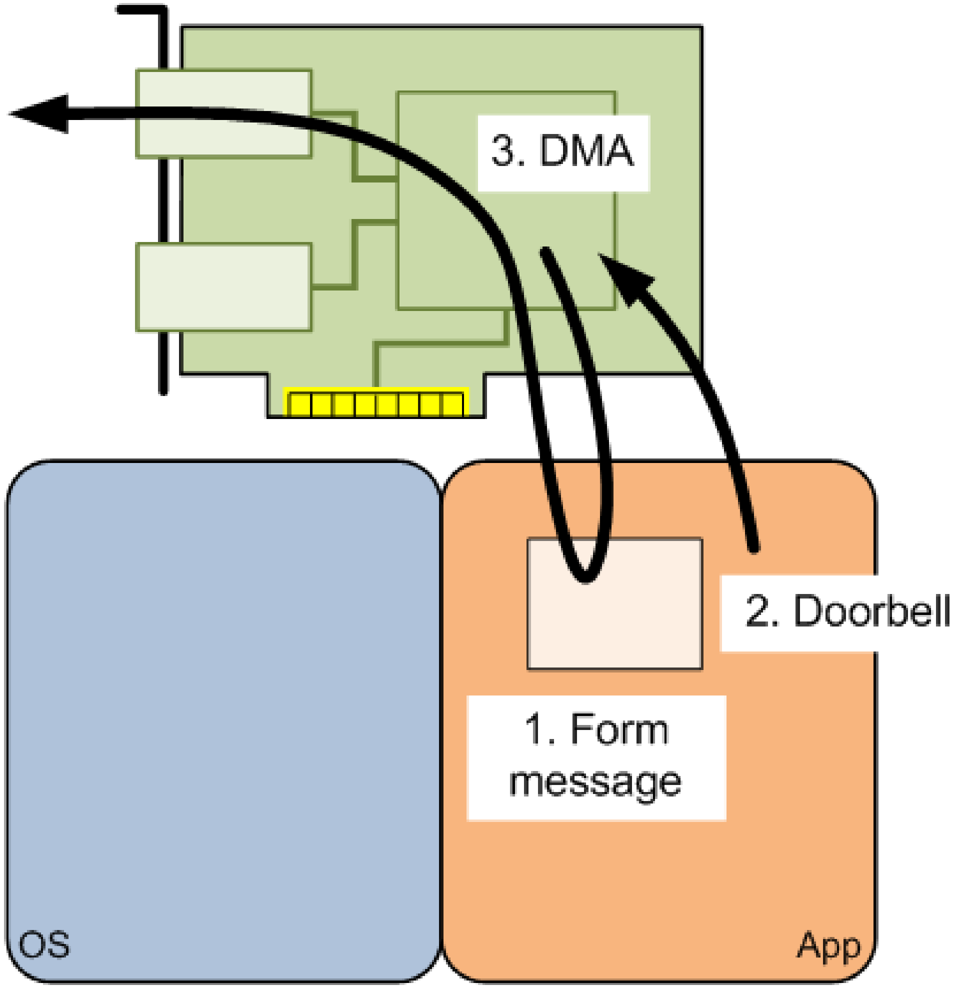
Kernel-bypass transmit



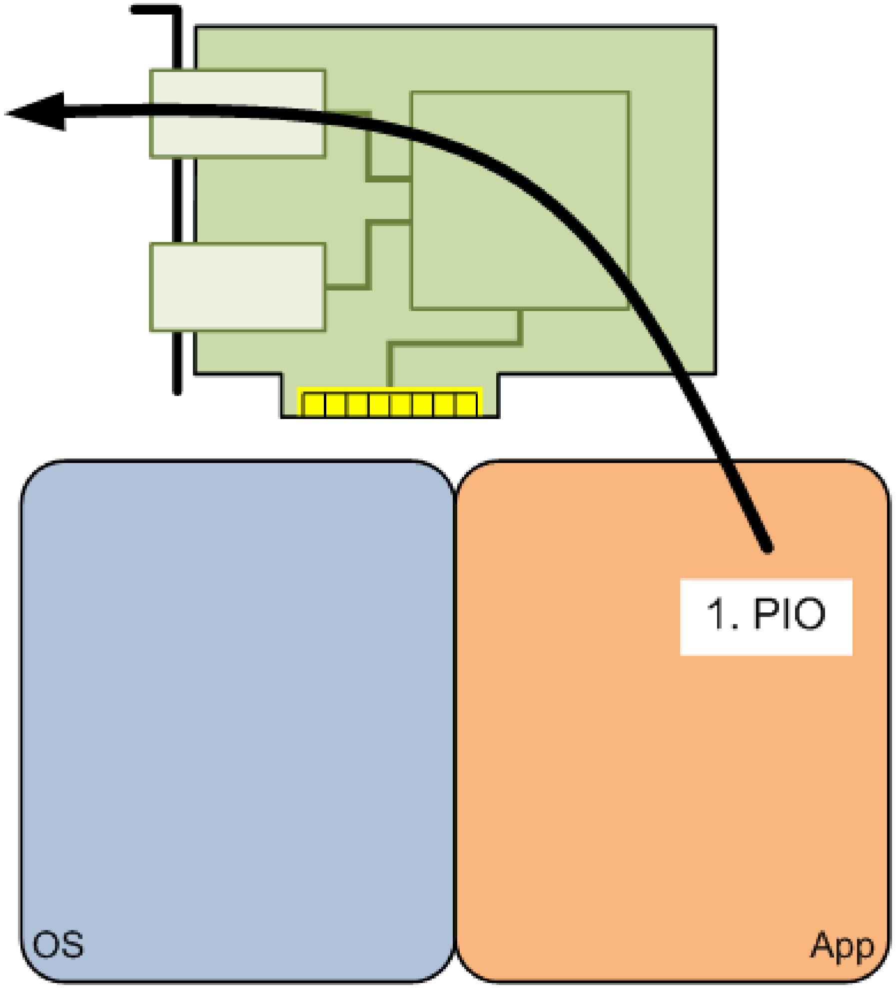
Kernel-bypass transmit



Kernel-bypass transmit



Kernel-bypass transmit – even faster

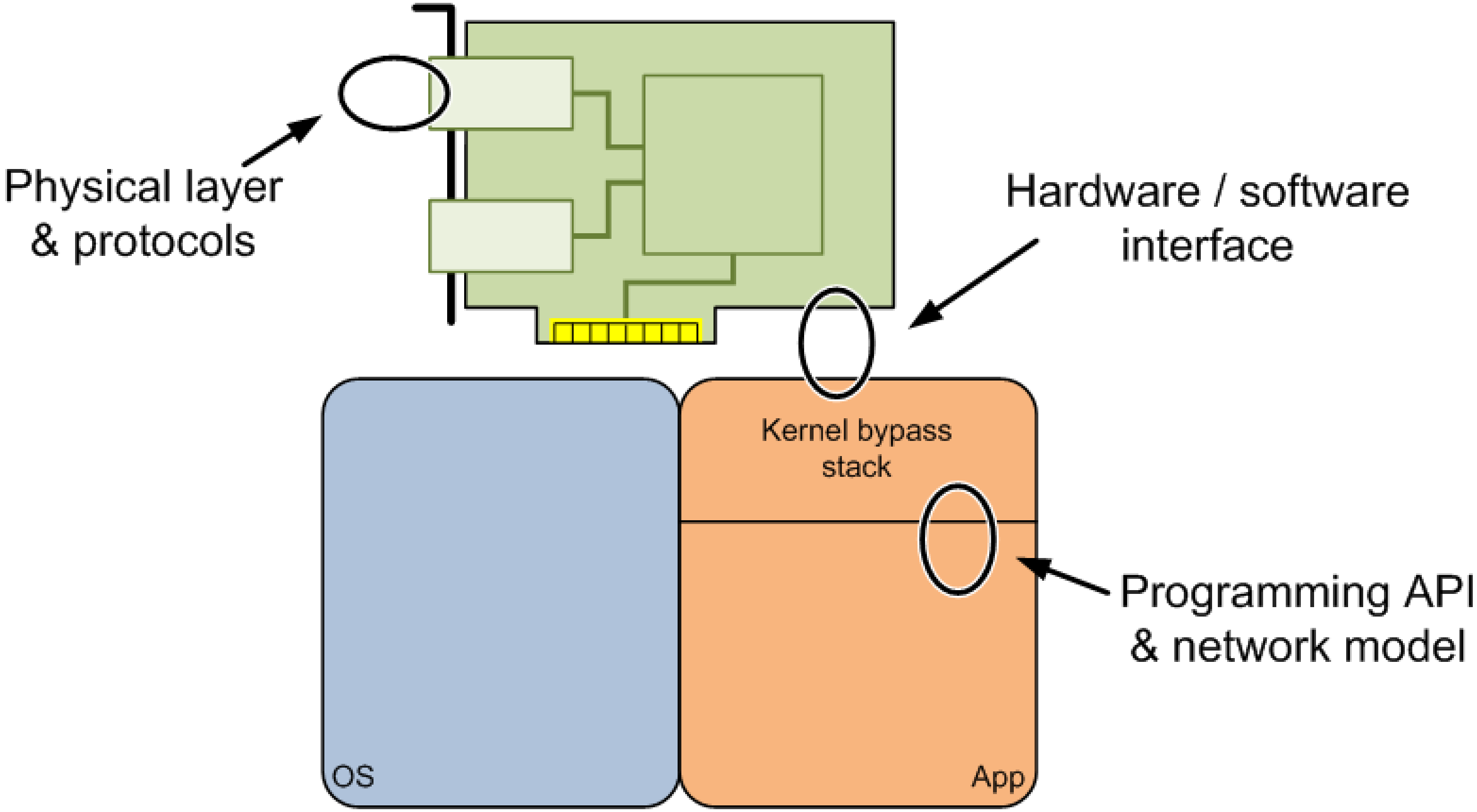


What does all of this cleverness achieve?

Better performance!

- Fewer CPU instructions for network operations
- Better cache locality
  - Faster response (lower latency)
  - Higher throughput (higher bandwidth/message rate)

- Reduced contention between threads
  - Better core scaling
  - Reduced latency jitter



Physical layer & protocols

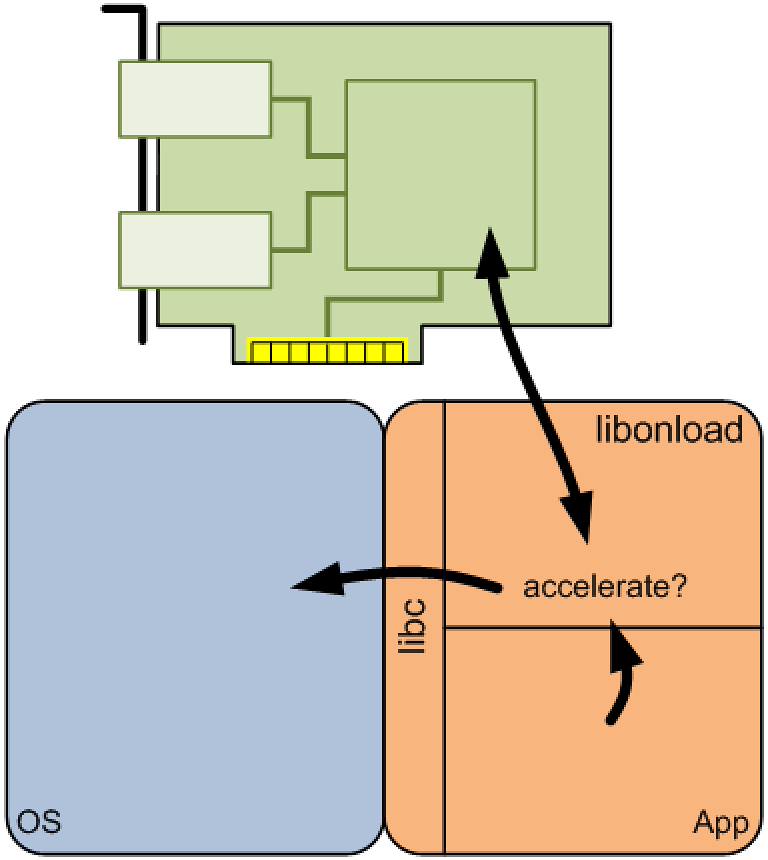
Hardware / software interface

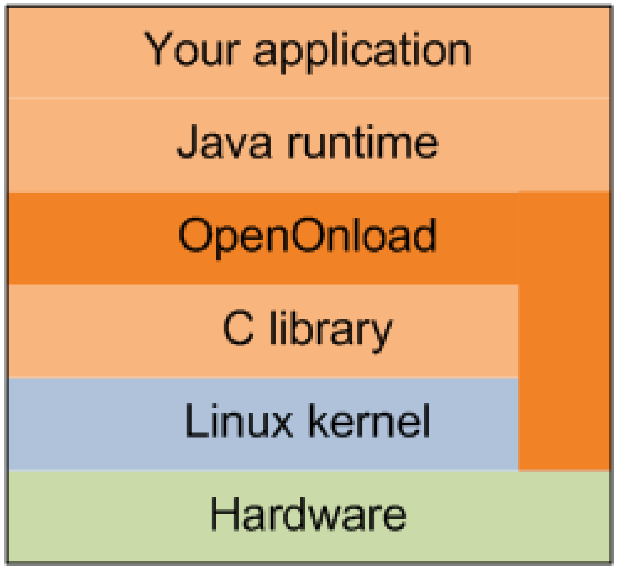
Programming API & network model

Solarflare's OpenOnload

- Sockets acceleration using kernel bypass
- Standard Ethernet, IP, TCP and UDP
- Standard BSD sockets API
- Binary compatible with existing applications

OpenOnload intercepts network calls





Your application

Java runtime

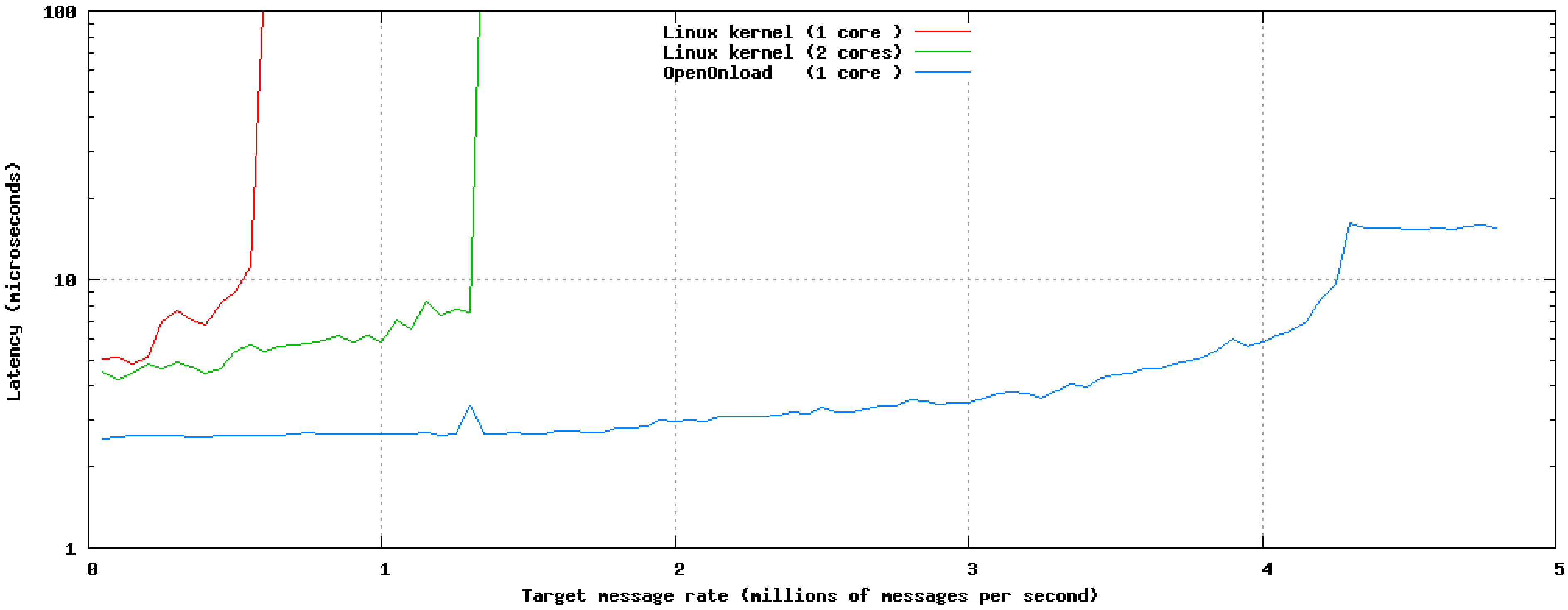
OpenOnload

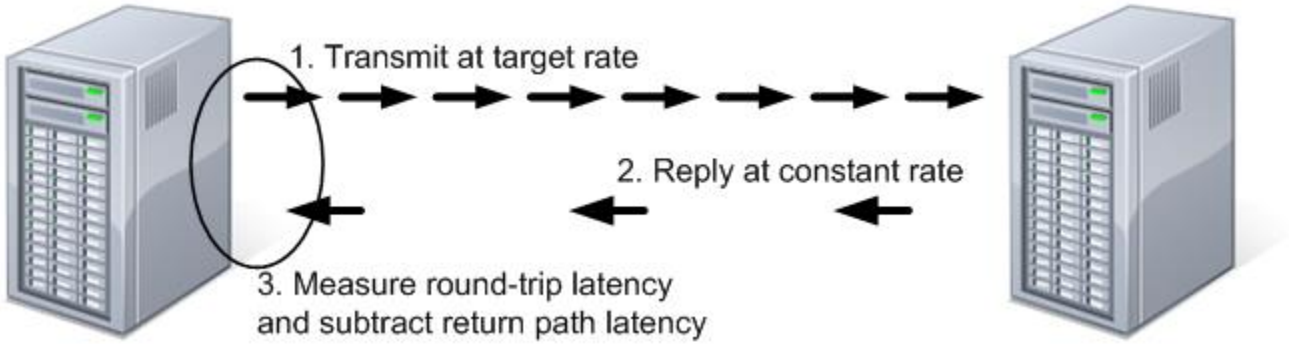
C library

Linux kernel

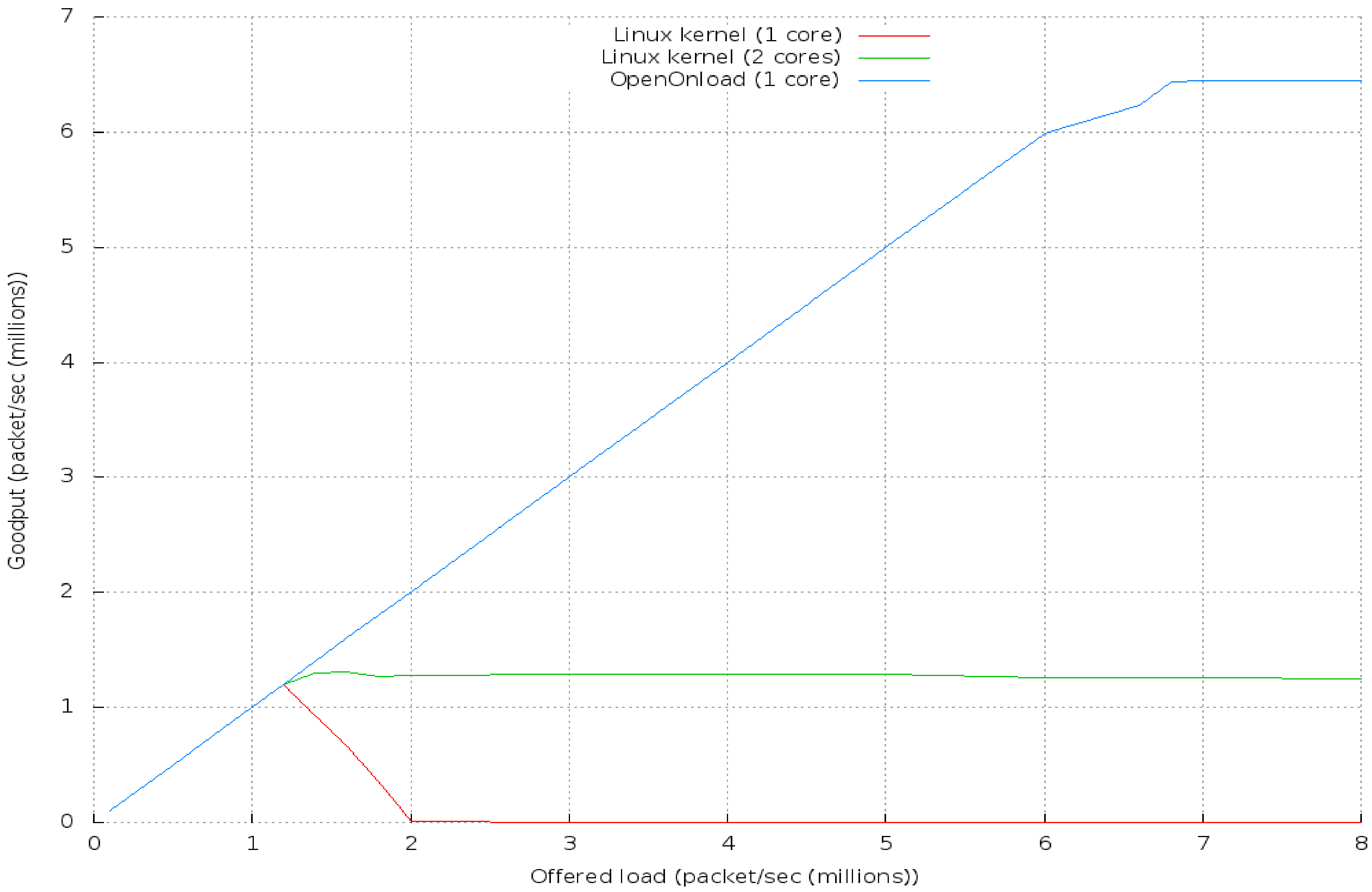
Hardware

Single thread throughput and latency

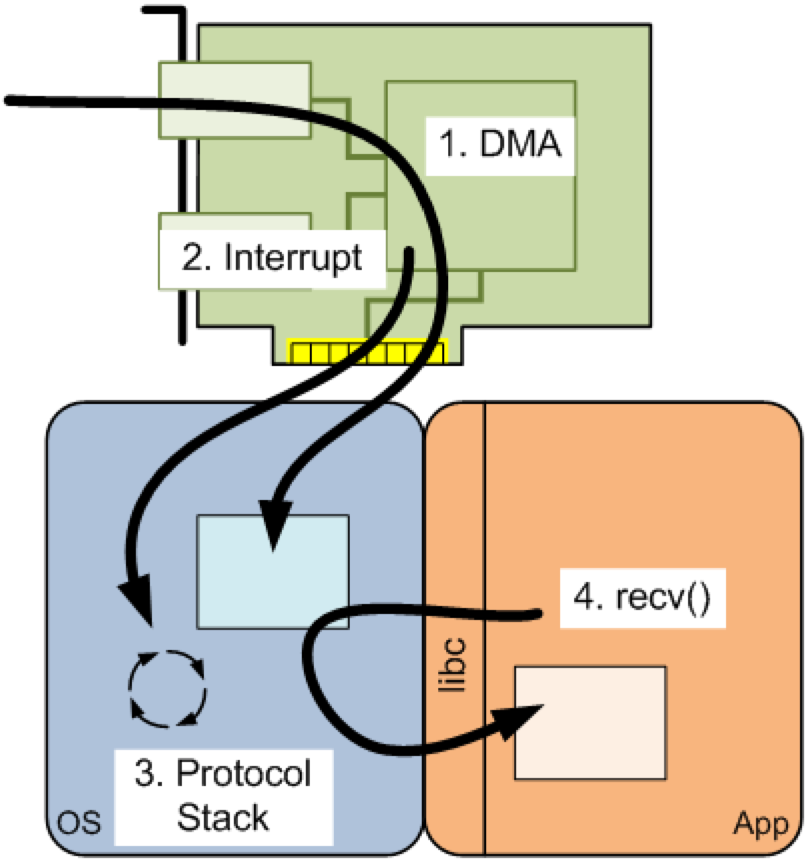




UDP receive throughput (small messages)



Kernel stack



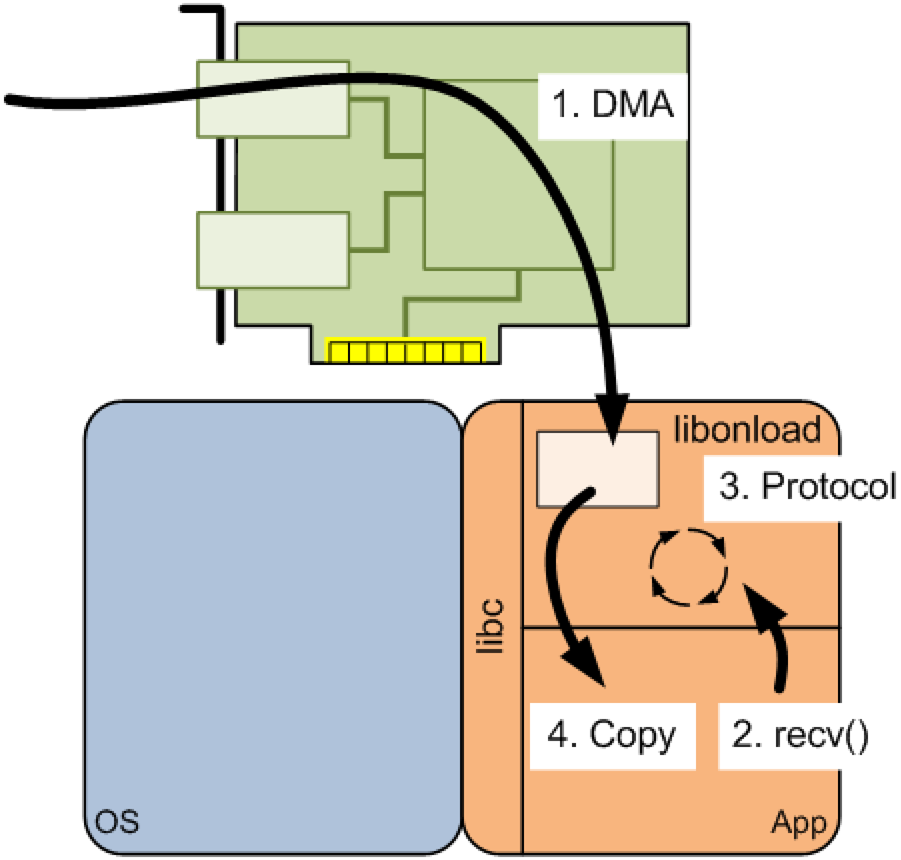
1. DMA

2. Interrupt

3. Protocol Stack

4. recv()

OpenOnload



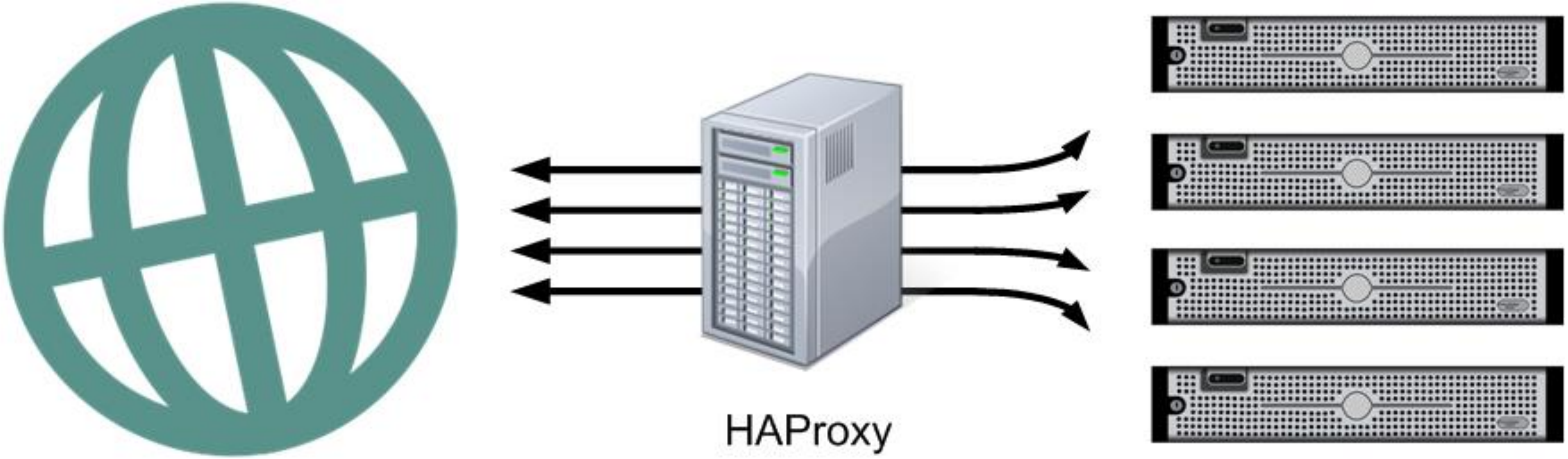
1. DMA

2. recv()

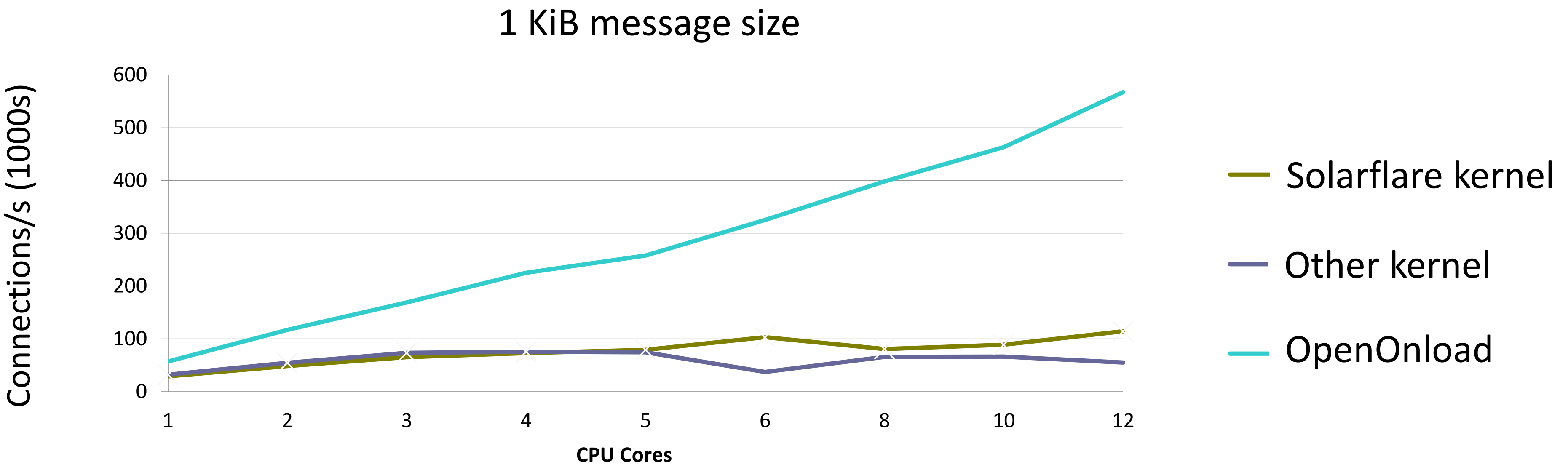
3. Protocol

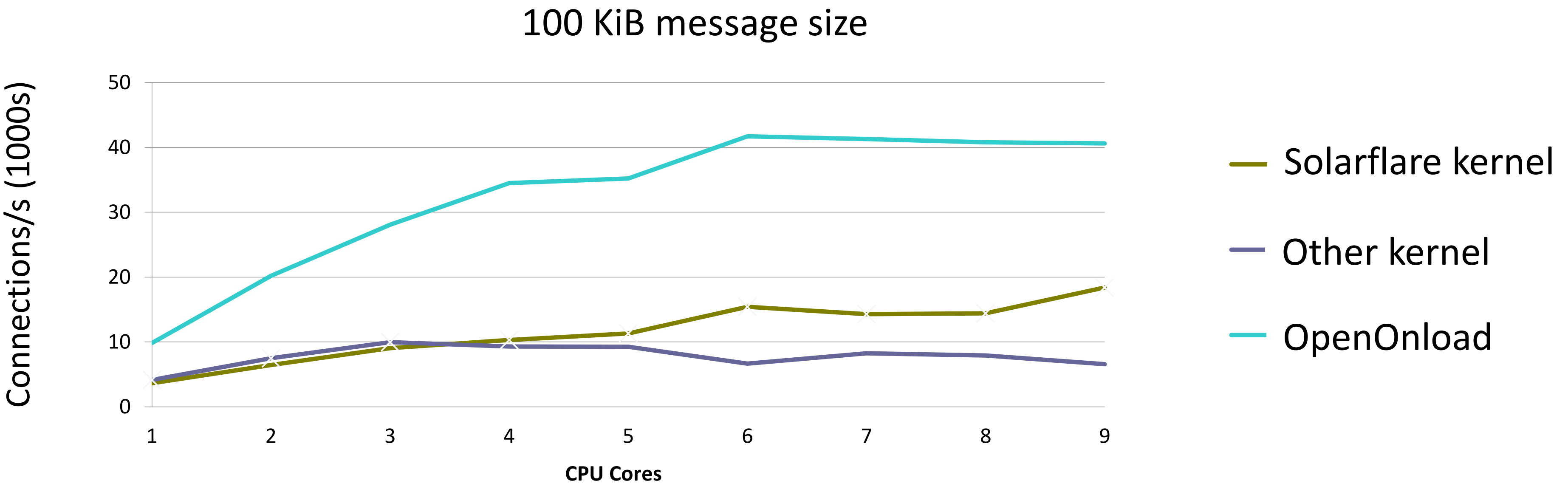
4. Copy

A much more challenging application



HAProxy performance and scaling

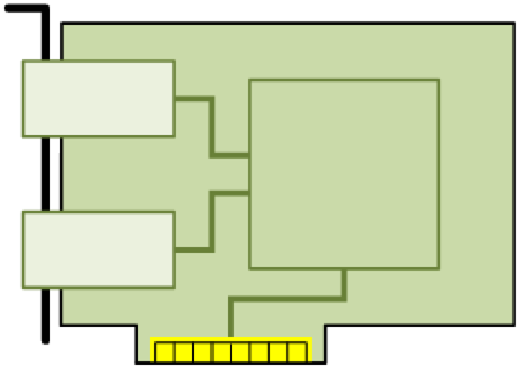


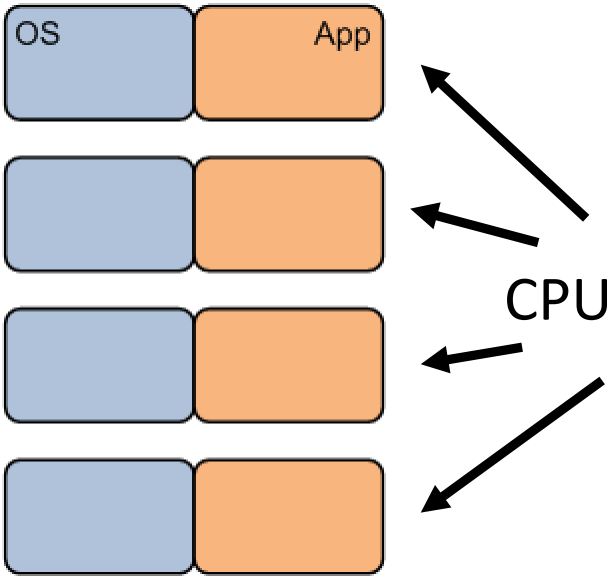


Why doesn’t performance scale when using the kernel stack?

Better question:

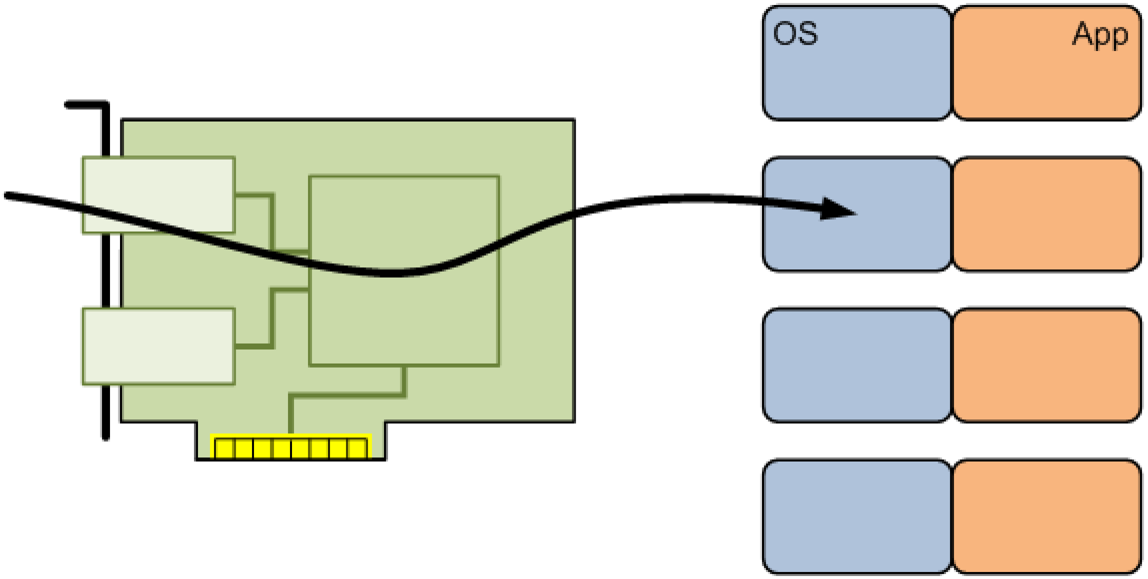
How come it scales as well as it does?

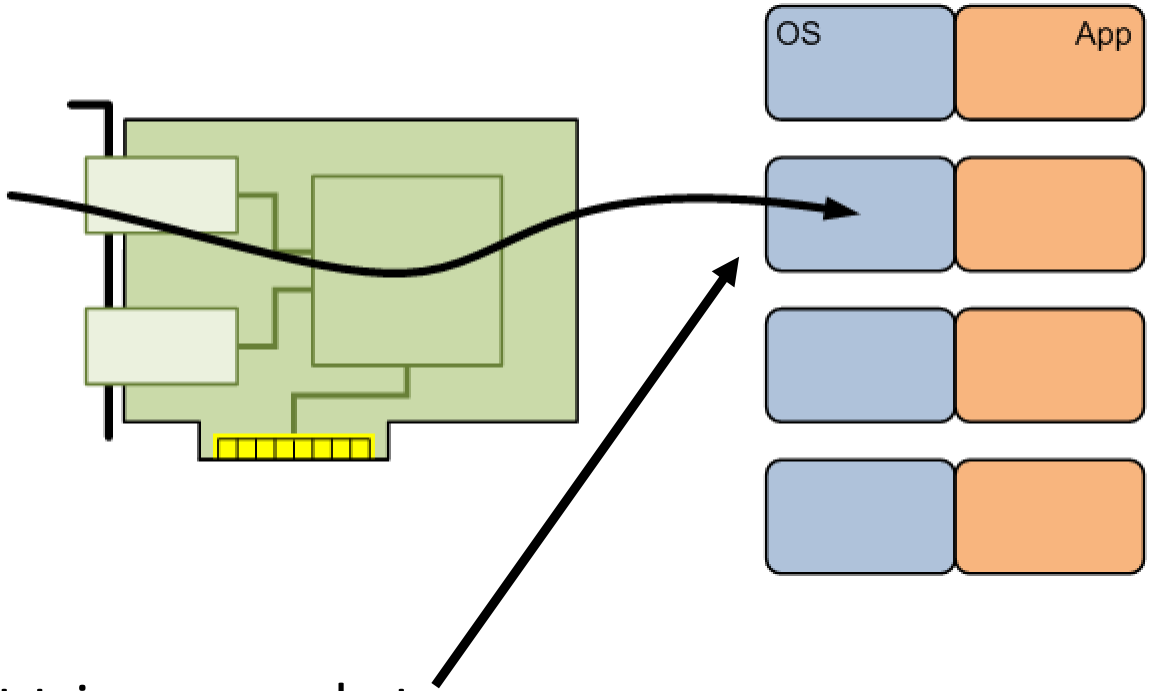




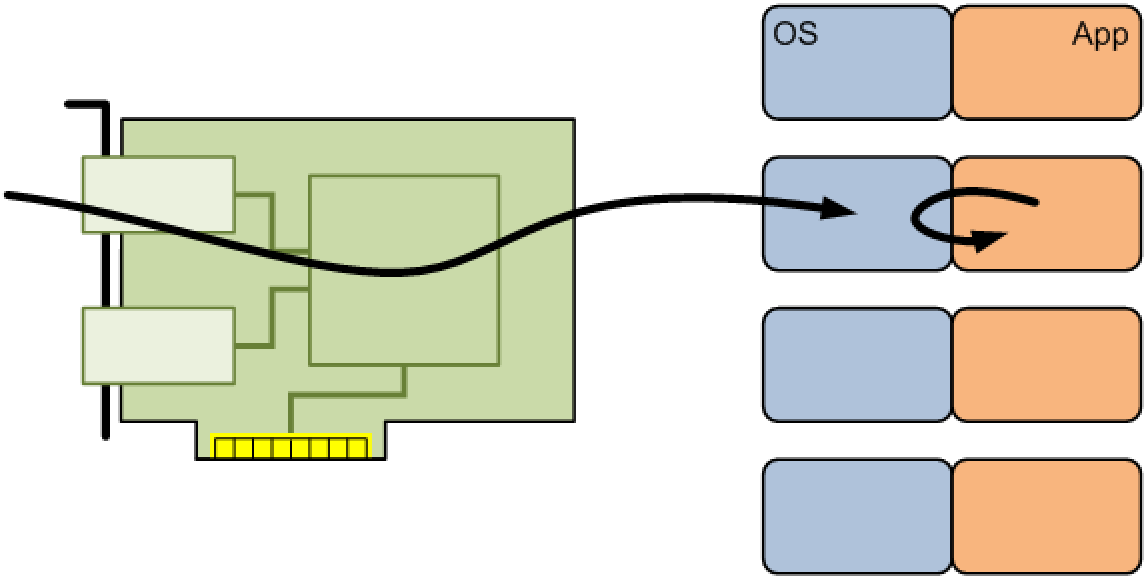
CPU cores

Received packet is delivered into memory (or L3 cache)



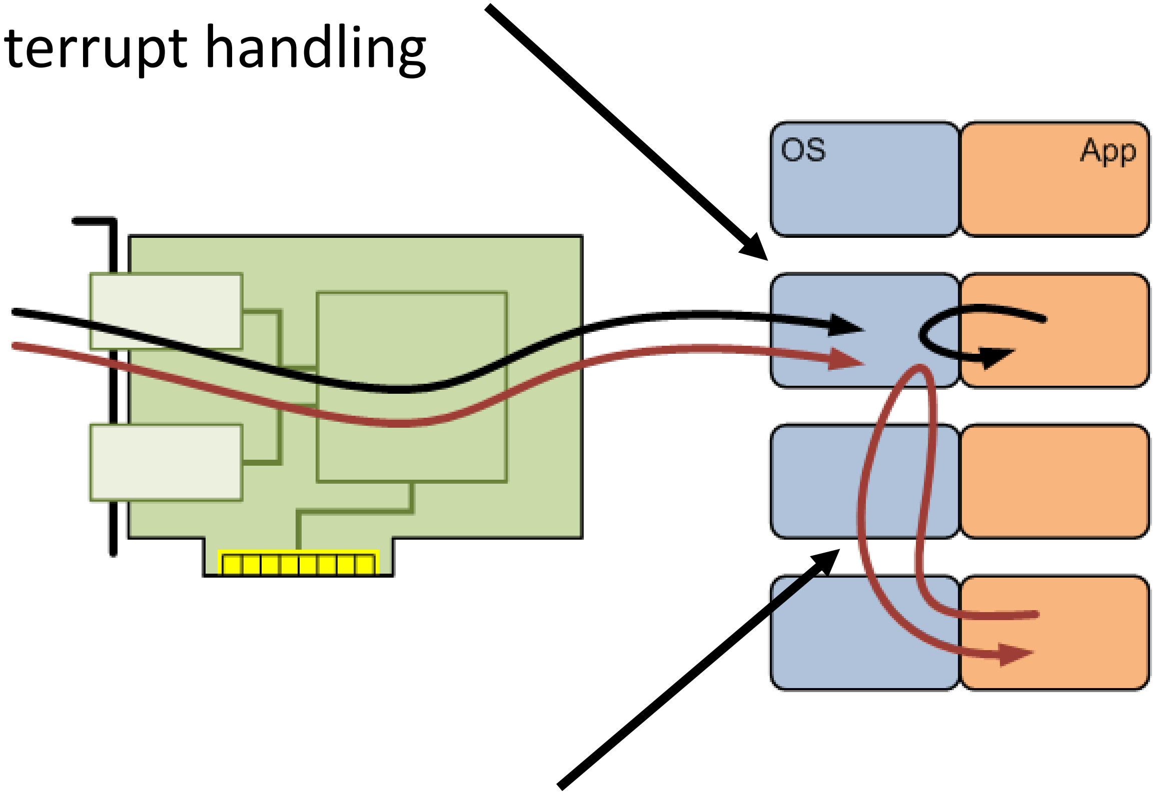


Interrupt triggers packet handling on this CPU core



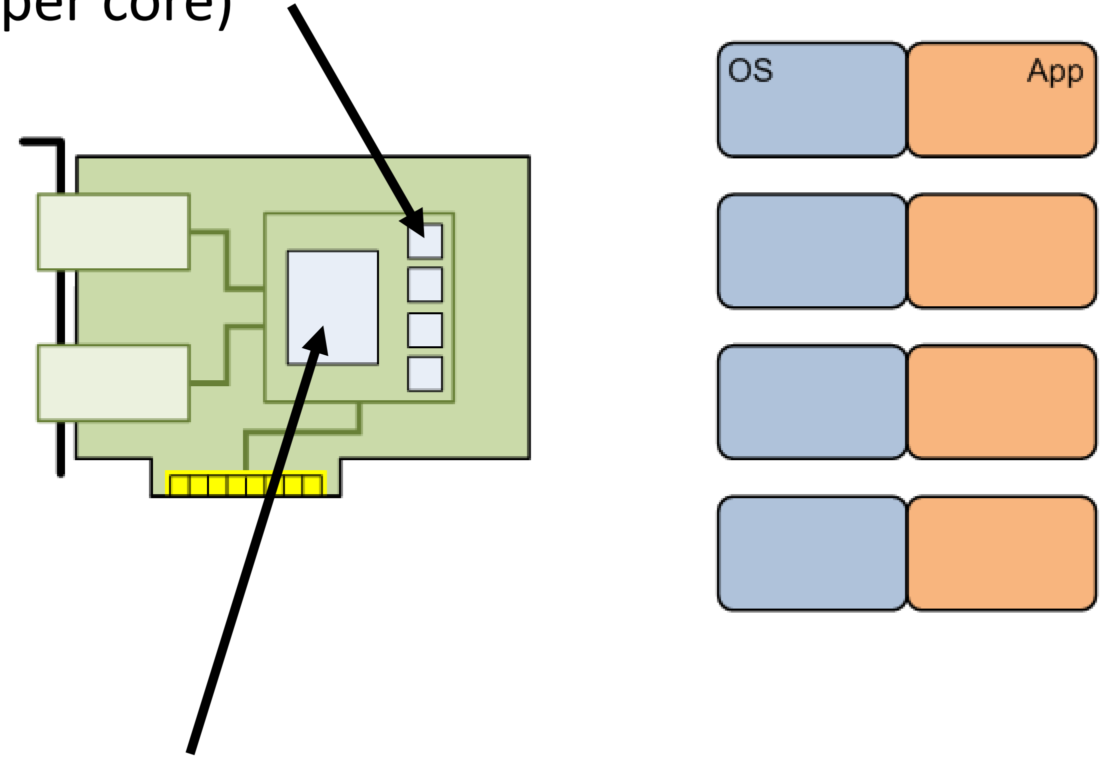
Application calls recv()

Single core bottleneck for interrupt handling

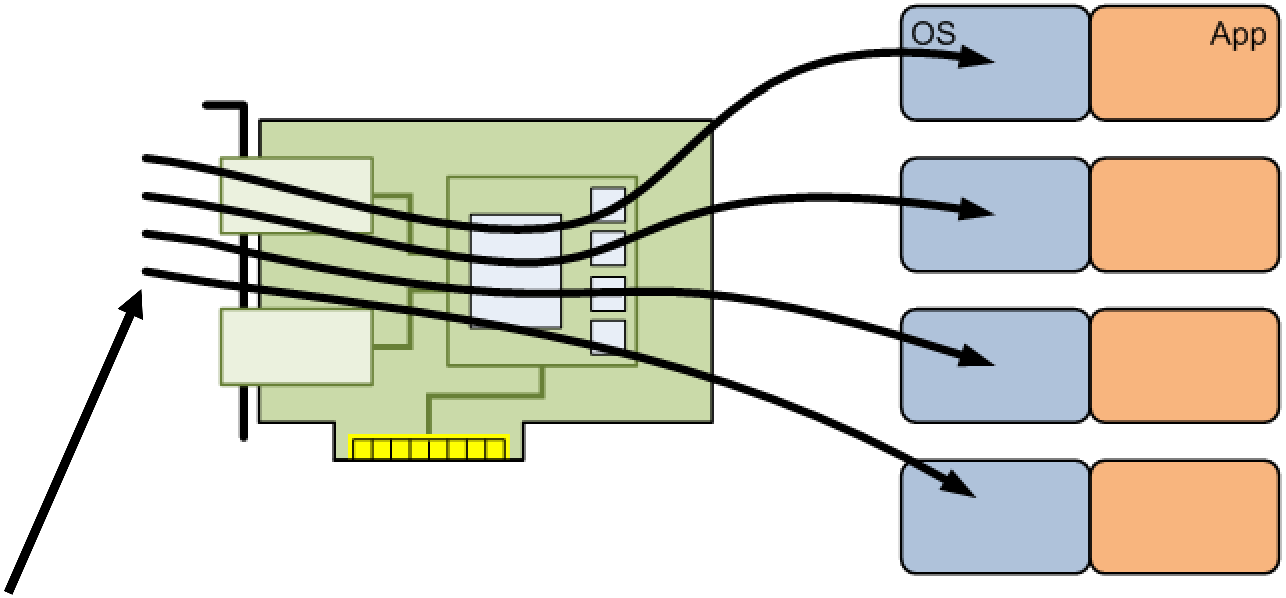


Socket state pulled from one cache to another; **Inefficient**

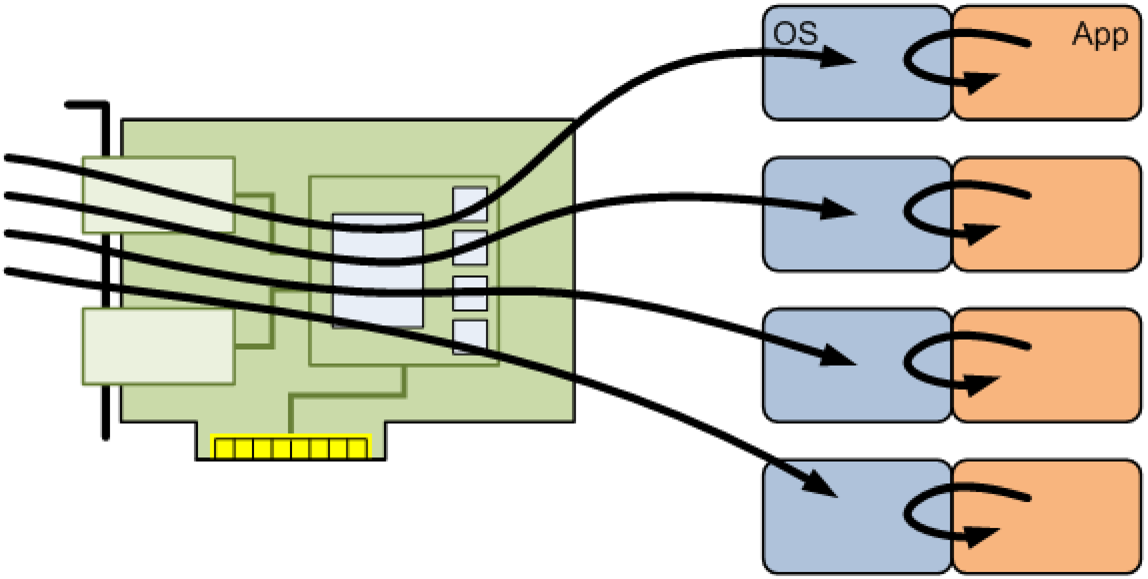
Multiple receive channels
(up to one per core)



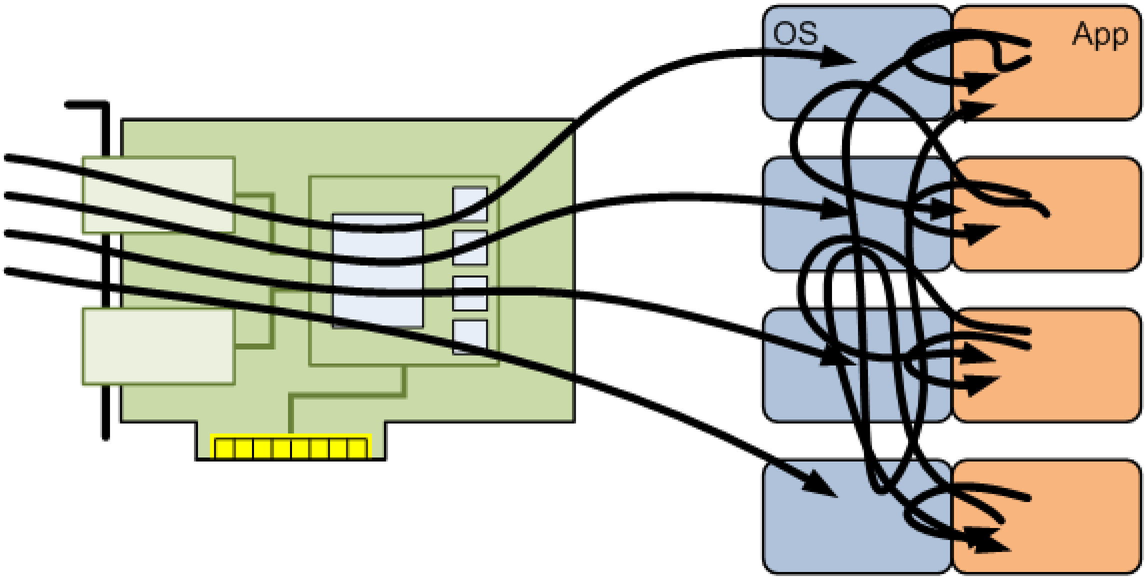
`channel_id = hash(4tuple) % n_cores;`



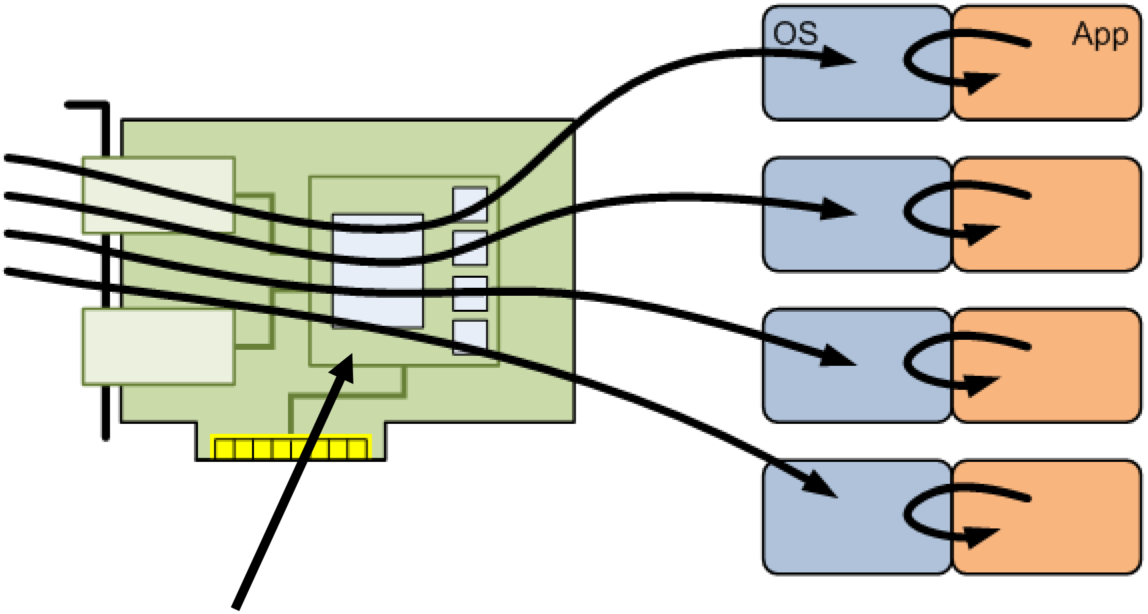
Multiple flows



Hopefully!



Usually



```
channel_id, n = lookup(4tuple);
if( n > 1 )
    channel_id += hash(4tuple) % n_cores;
```

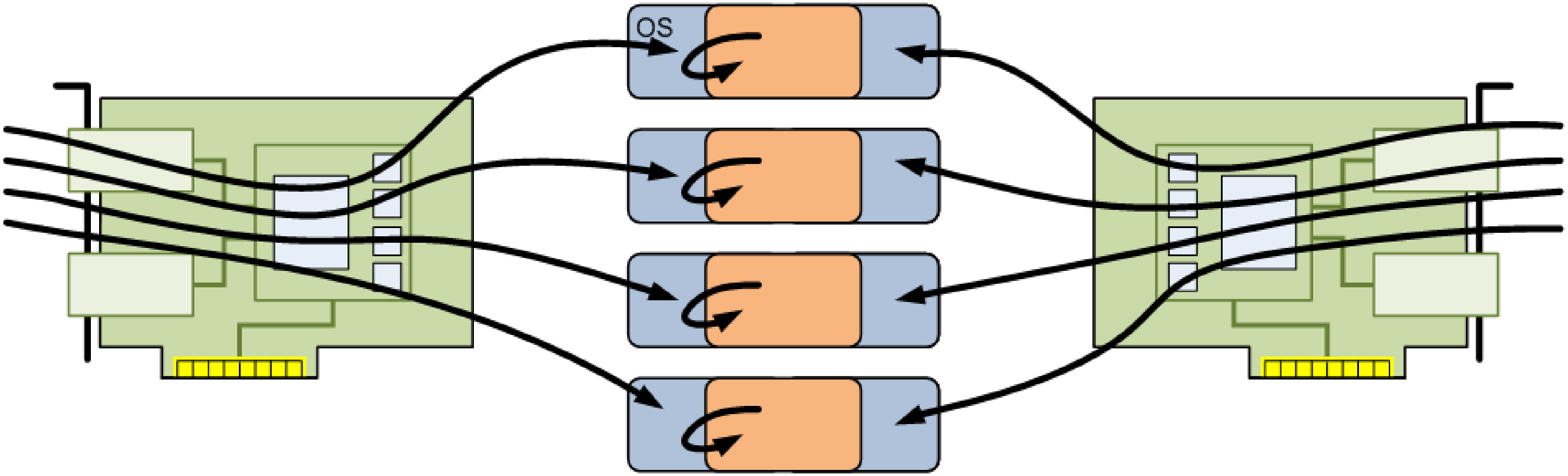
SO_REUSEPORT to the rescue

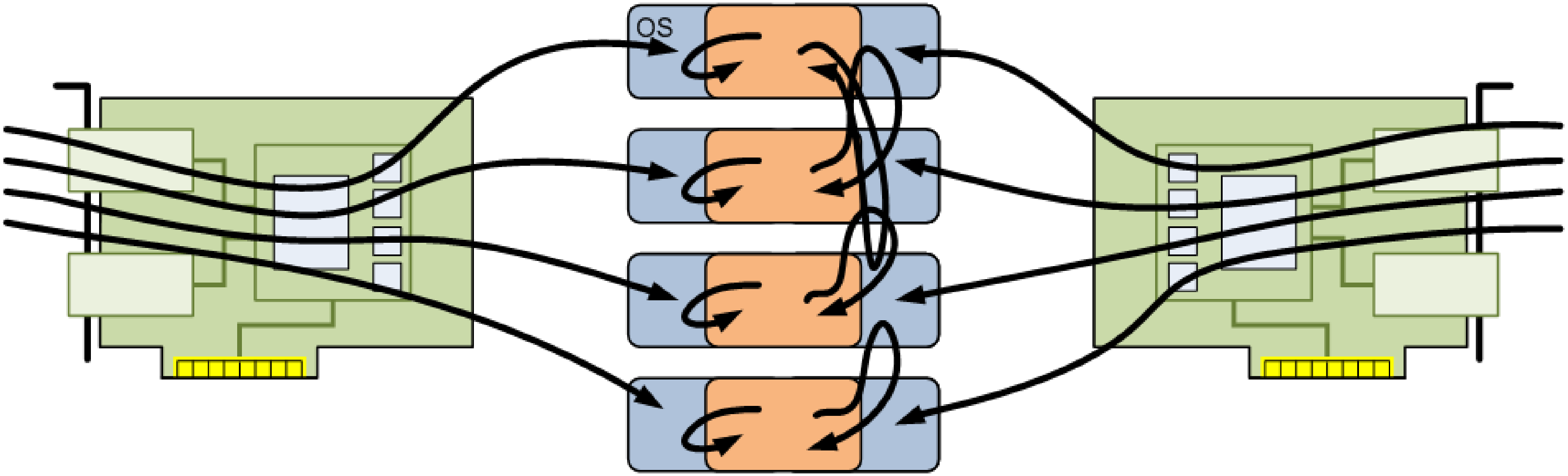
- Multiple listening sockets on the same TCP port
  - One listening socket per worker thread
  - Each gets a subset of incoming connection requests
  - New connections go to the worker running on the core that the flow hashes to

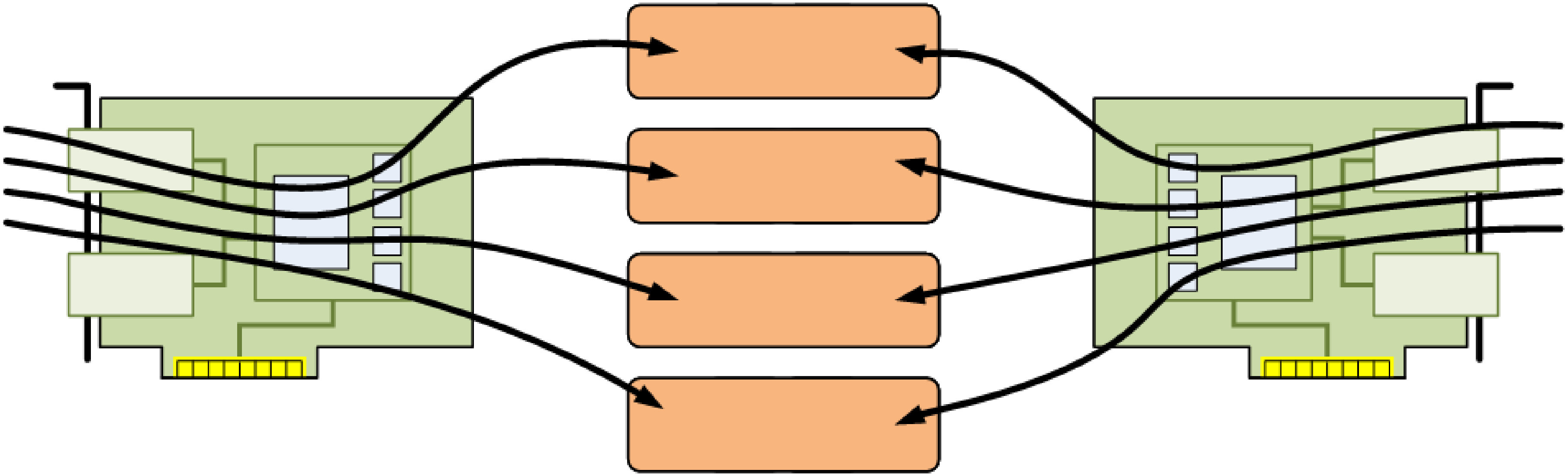
- Connection establishment scales with the number of workers

- Received packets are delivered to the ‘right’ core

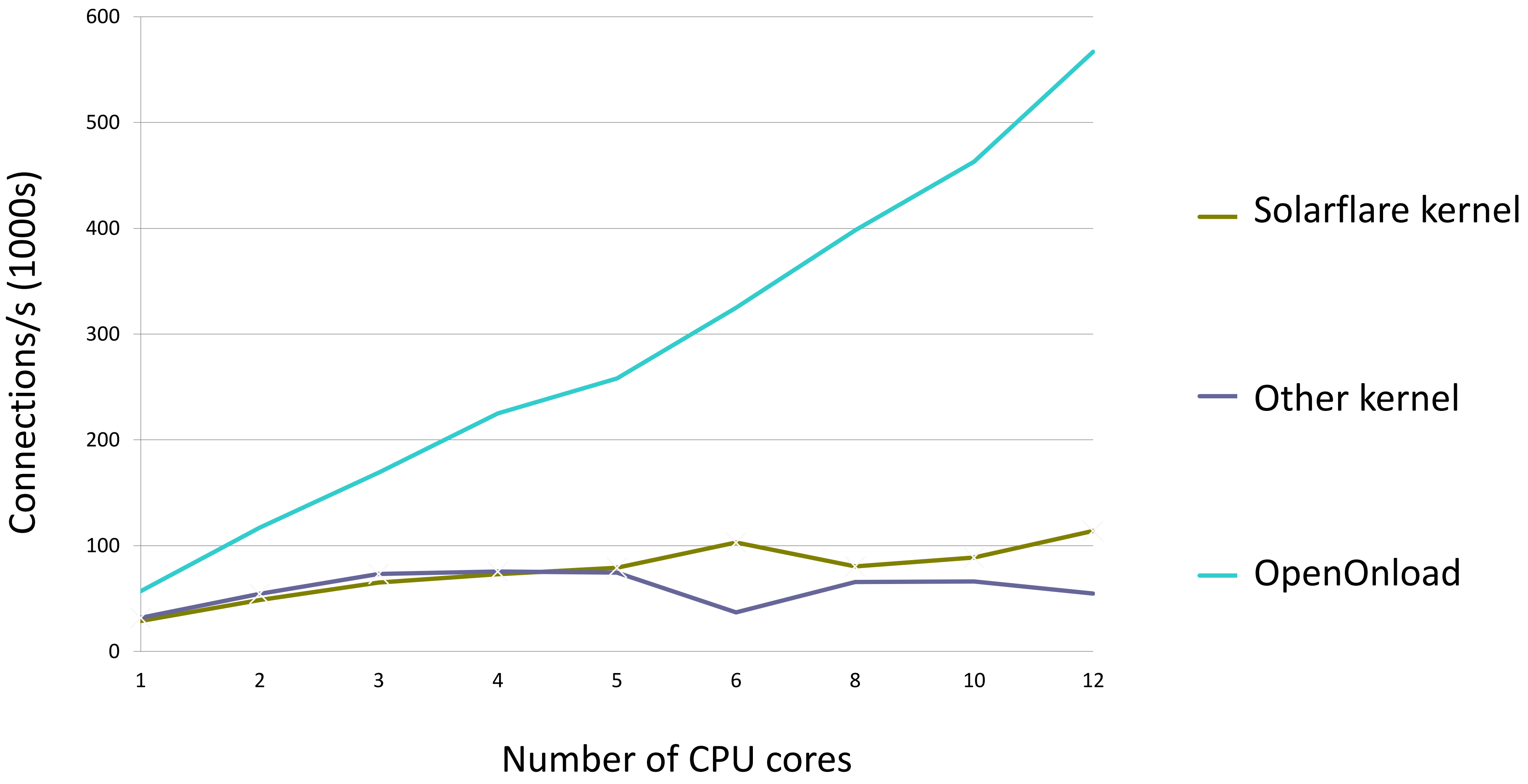
Problem solved?


OS

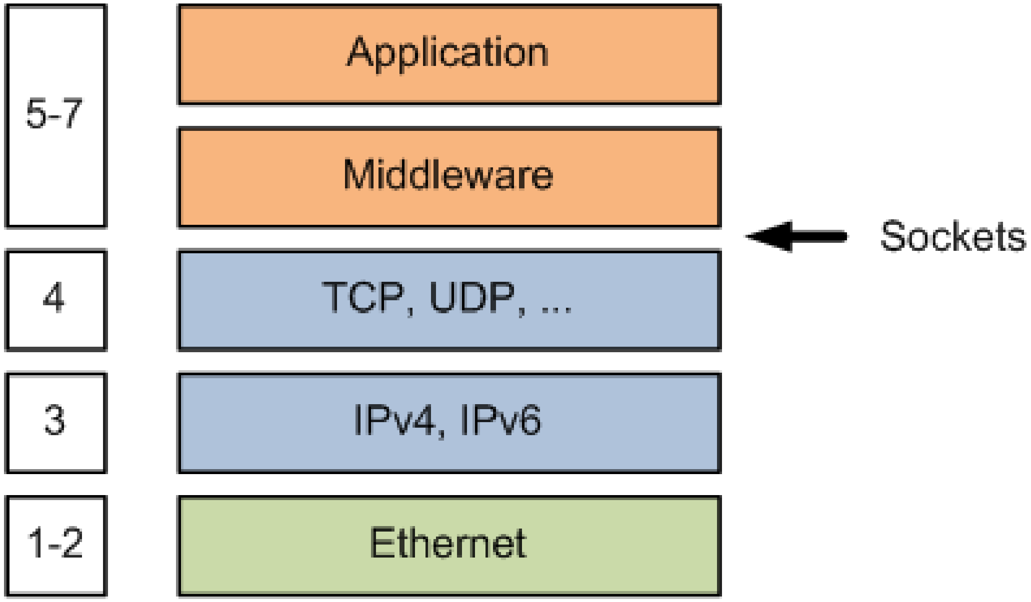

OS



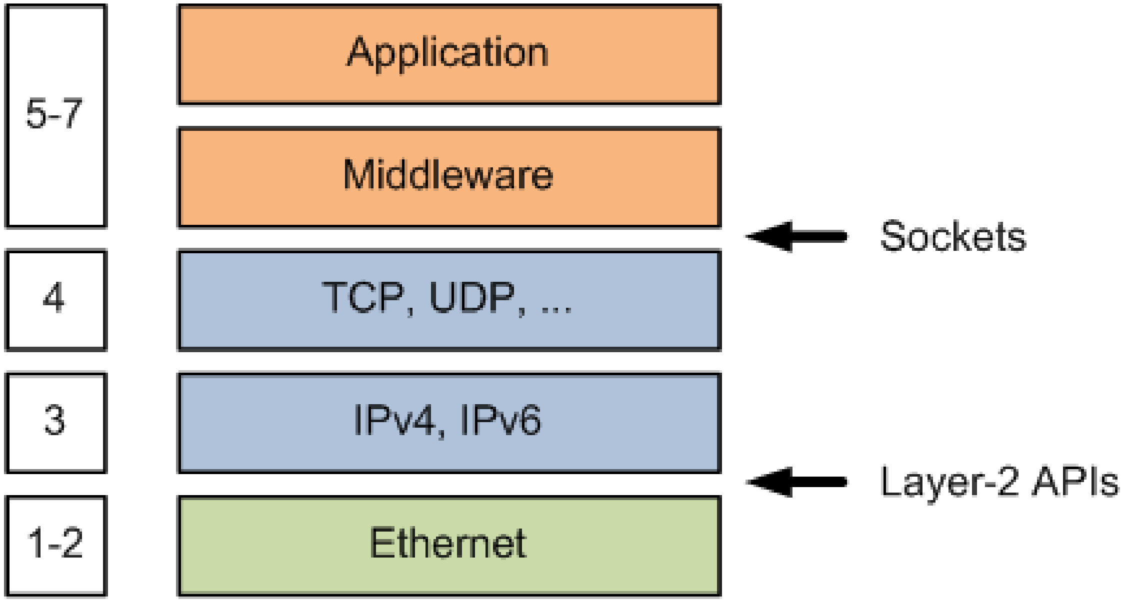
1 KiB message size



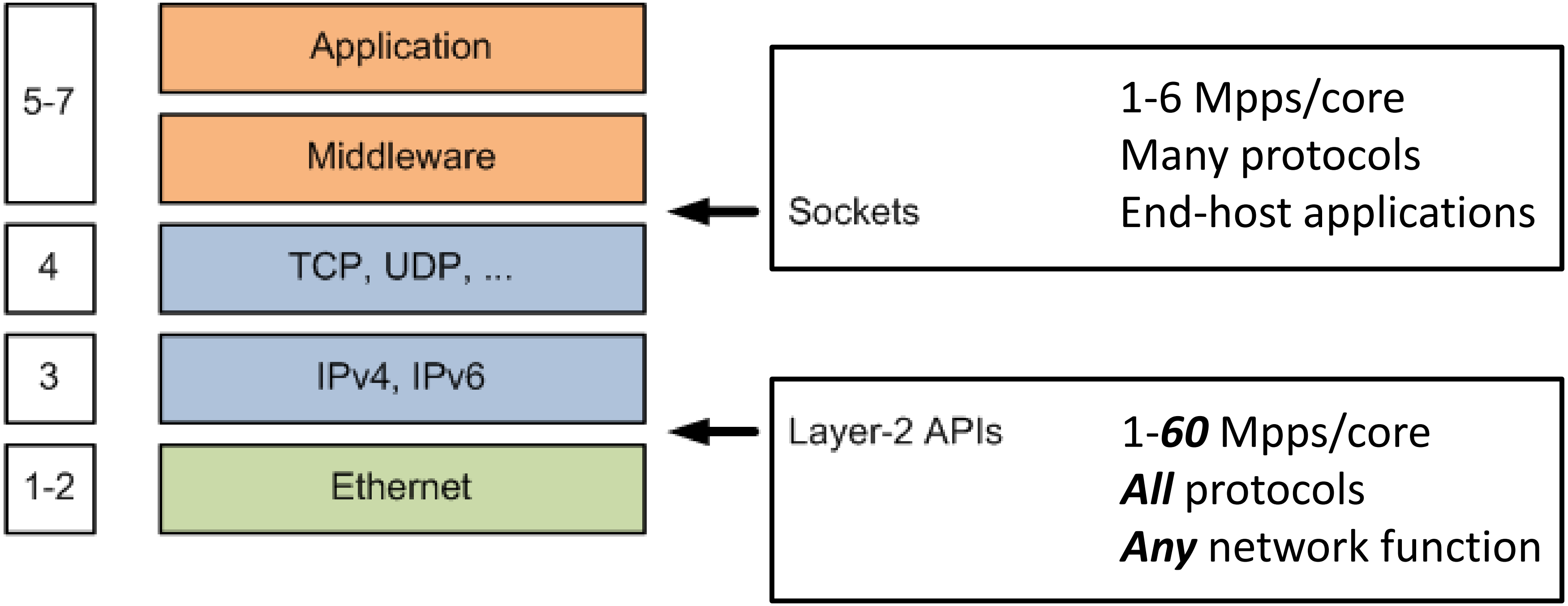
So much for sockets



Let’s get closer to the metal…



Layer-2 APIs



Sockets

1-6 Mpps/core
Many protocols
End-host applications

Layer-2 APIs

1-***60*** Mpps/core
All protocols
Any network function

60 million pkt/s?

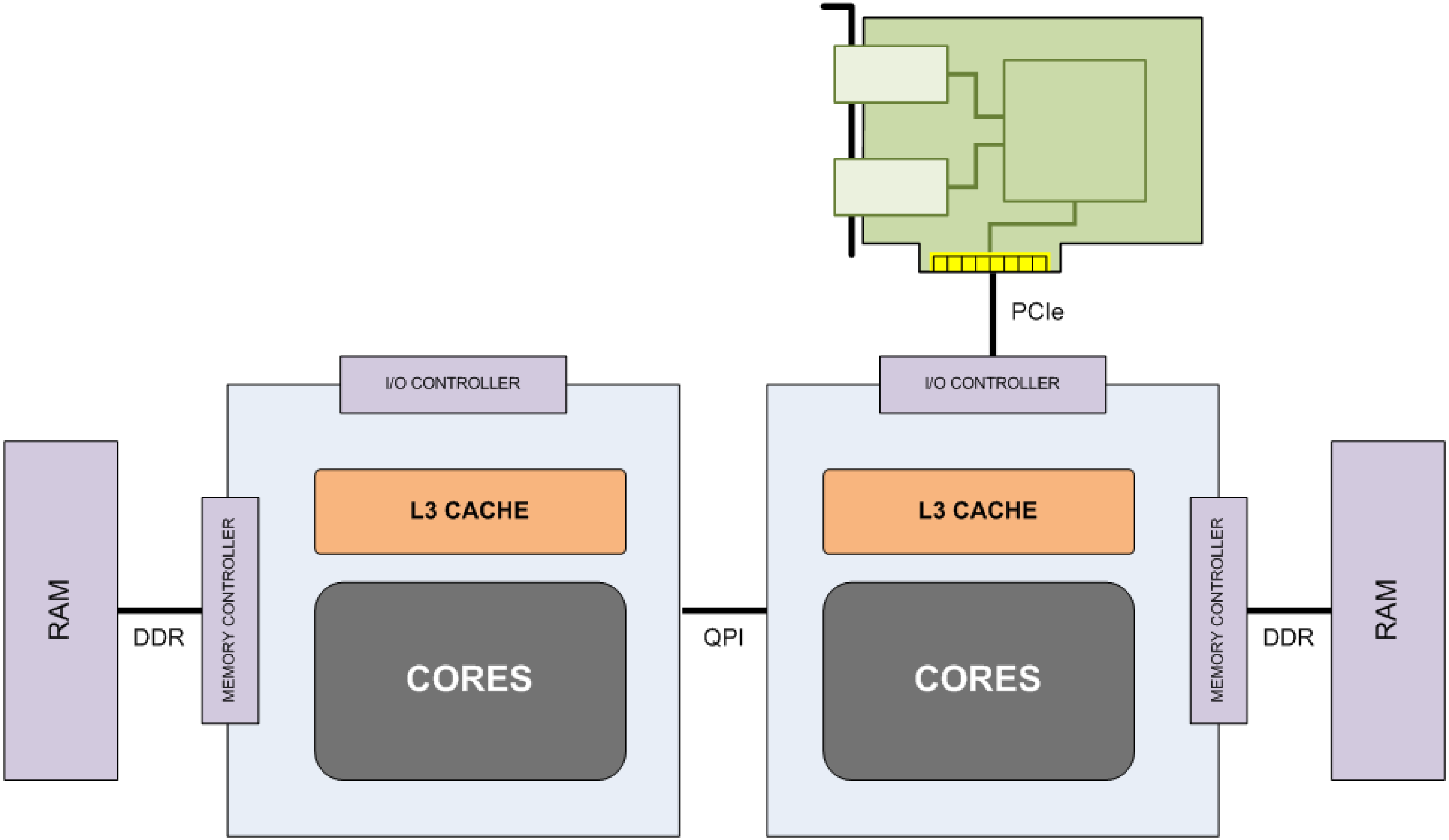
17 ns

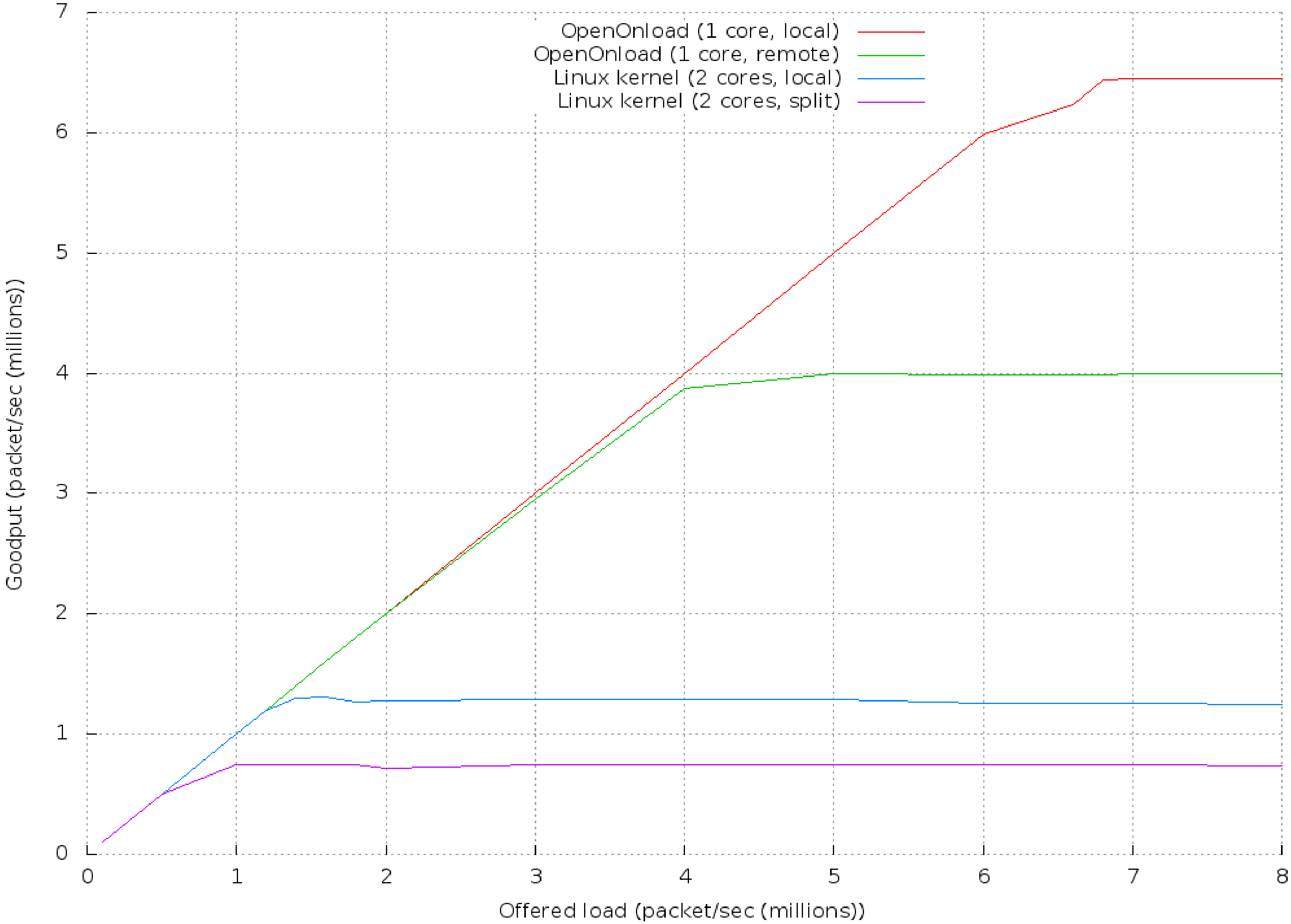
Some tips for achieving really fast networking…

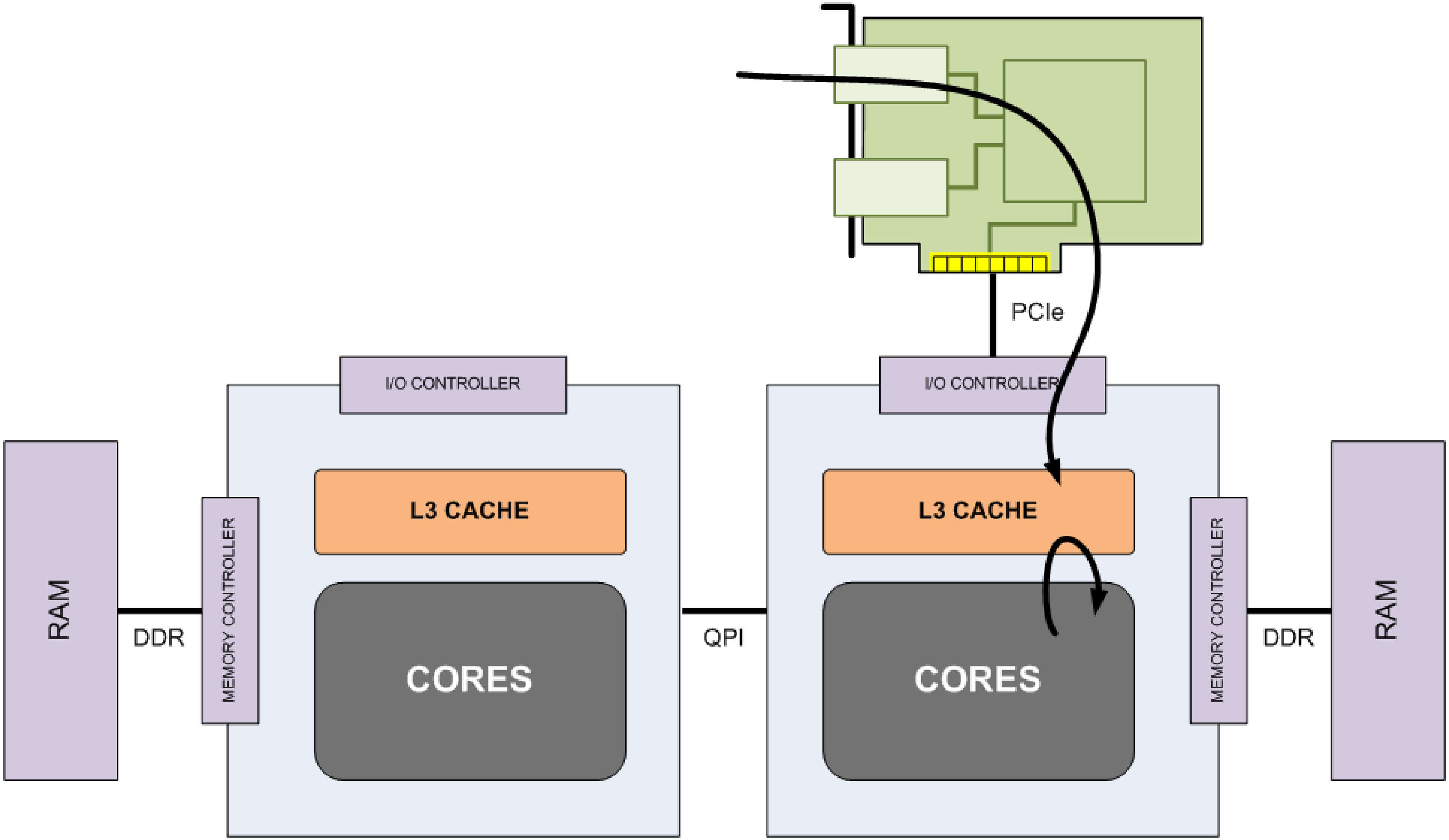
Tip 1. The faster your application is already, the more speedup you'll get from kernel bypass

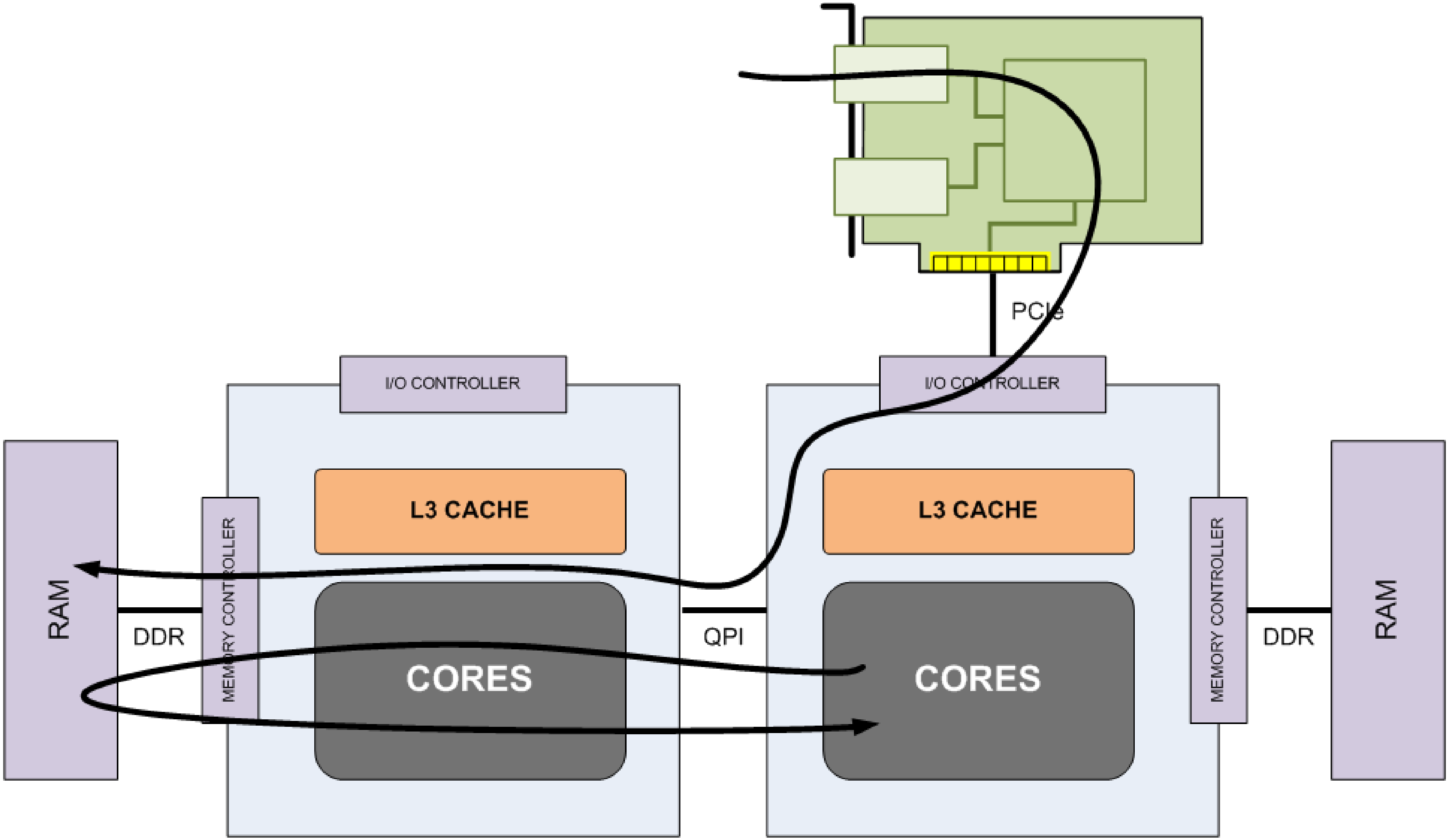
(This is just Ahmdal's law)

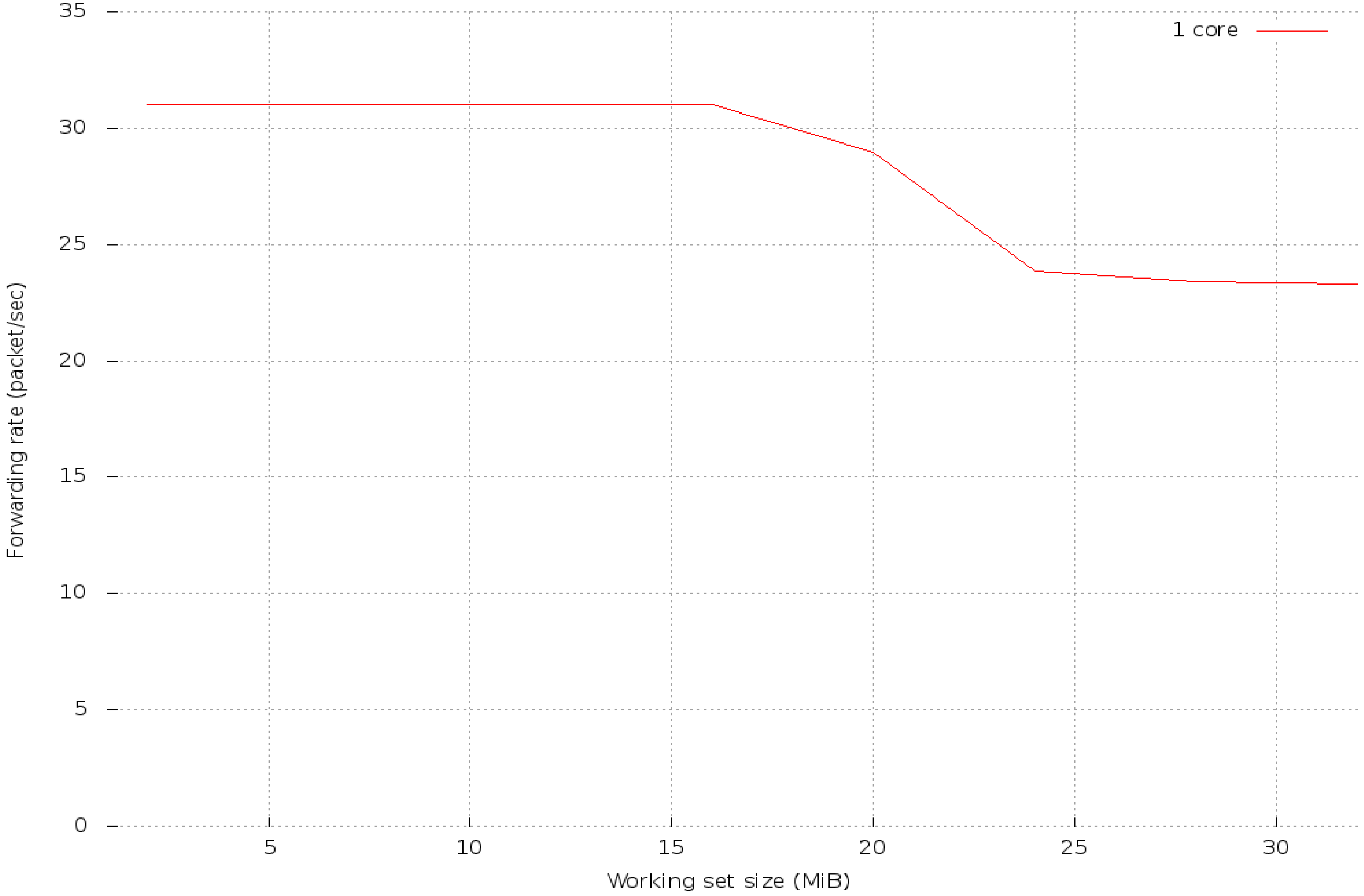
Tip 2. NUMA locality applies doubly to I/O devices


PCIe
I/O CONTROLLER
I/O CONTROLLER
L3 CACHE
L3 CACHE
CORES
CORES
RAM
DDR
MEMORY CONTROLLER
QPI
MEMORY CONTROLLER
DDR
RAM


OpenOnload (1 core, local)
OpenOnload (1 core, remote)
Linux kernel (2 cores, local)
Linux kernel (2 cores, split)
Goodput (packet/sec (millions))
Offered load (packet/sec (millions))
0
1
2
3
4
5
6
7
8


PCIe
I/O CONTROLLER
I/O CONTROLLER
RAM
DDR
MEMORY CONTROLLER
L3 CACHE
CORES
QPI
L3 CACHE
CORES
MEMORY CONTROLLER
DDR
RAM


PCIe
I/O CONTROLLER
I/O CONTROLLER
L3 CACHE
L3 CACHE
RAM
DDR
MEMORY CONTROLLER
CORES
QPI
CORES
MEMORY CONTROLLER
DDR
RAM



35
30
25
20
15
10
5
0
Forwarding rate (packet/sec)
5
10
15
20
25
30
Working set size (MiB)
1 core

- DMA transfers will use the L3 cache if:
  - The targeted cache line is resident
  - Or if not then up to 10% of L3 is available for write-allocate

- Therefore
  - If you want consistent high performance, DMA buffers must be resident in L3 cache

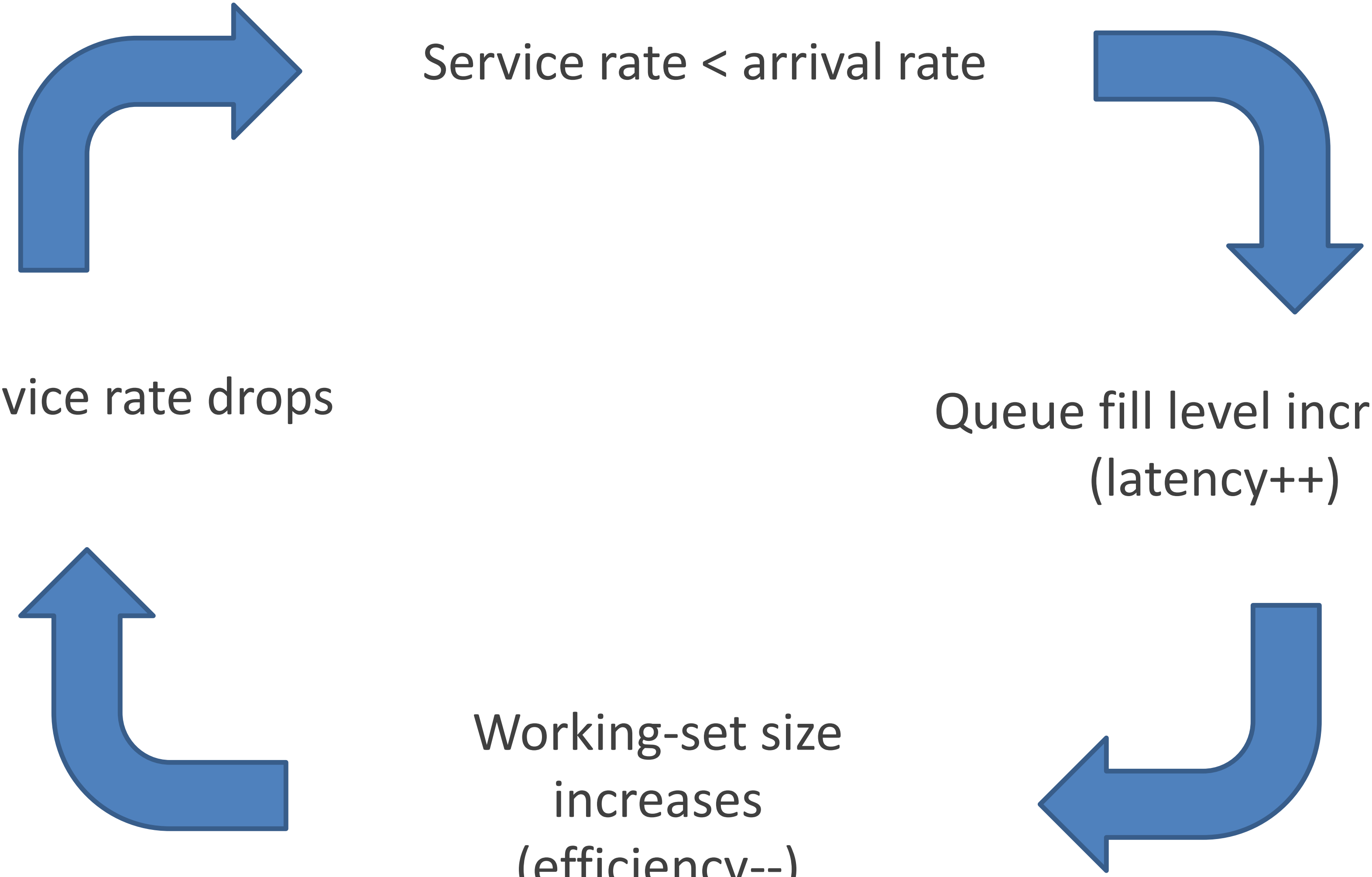
- To achieve that
  - Small set of DMA buffers recycled quickly
  - (Even if that means doing an extra copy)

Tip 3.
Queue management is *critical*

Queues exist mostly to handle mismatch between *arrival rate* and *service rate*

- Buffers in switches and routers
- Descriptor rings in network adapters
- Socket send and receive buffers
- Shared memory queues, locks etc.
- Run queue in the kernel task scheduler

What happens when queues start to fill?

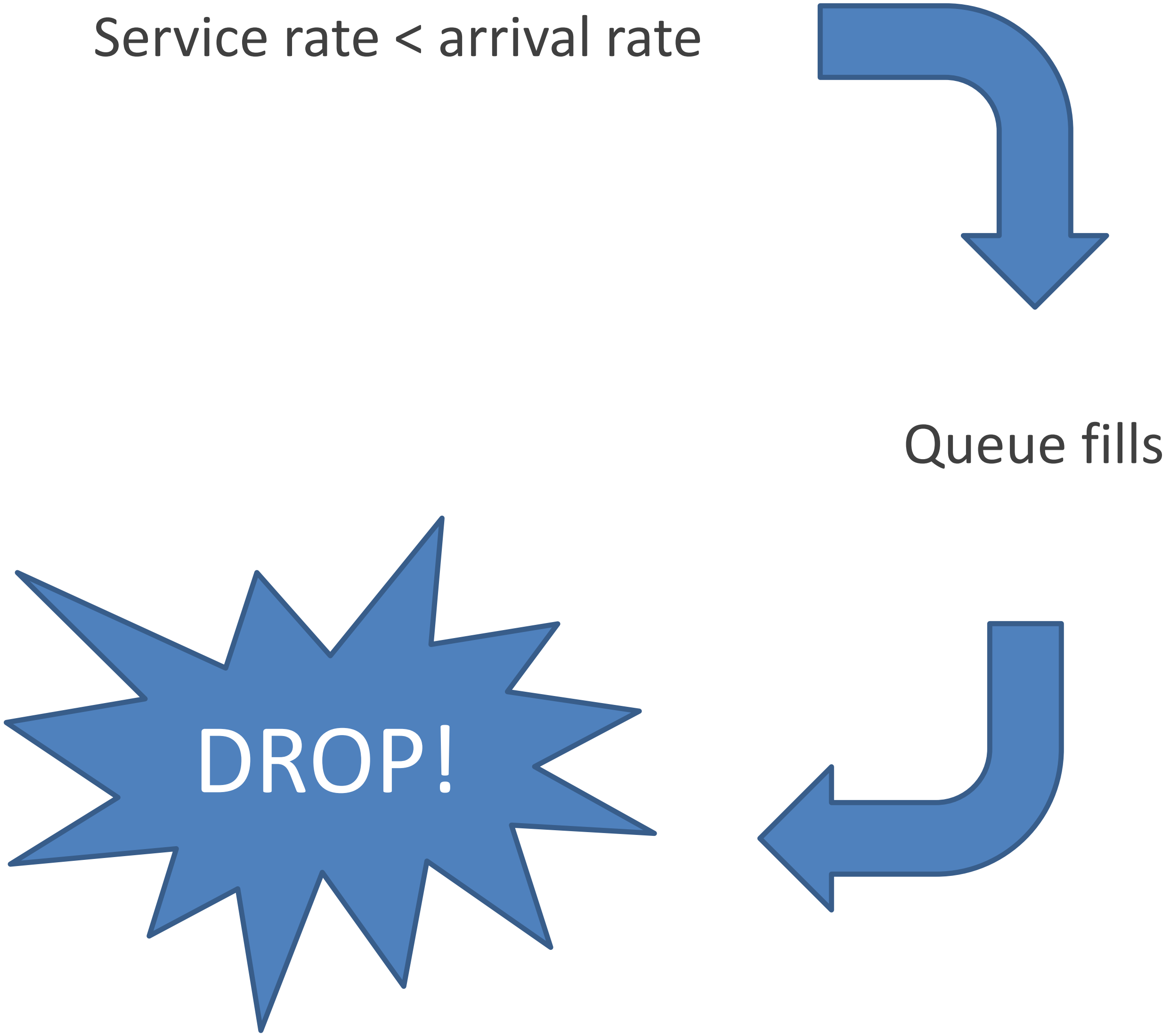


Service rate < arrival rate

Queue fill level increases
(latency++)

Working-set size
increases
(efficiency--)

Service rate drops



Service rate < arrival rate

Queue fills

DROP!

Drops are bad, m’kay

Make SO_RCVBUF bigger!

Dilemma!

Small buffers:

Necessary for stable performance when overloaded

Large buffers:

Necessary for absorbing bursts without loss

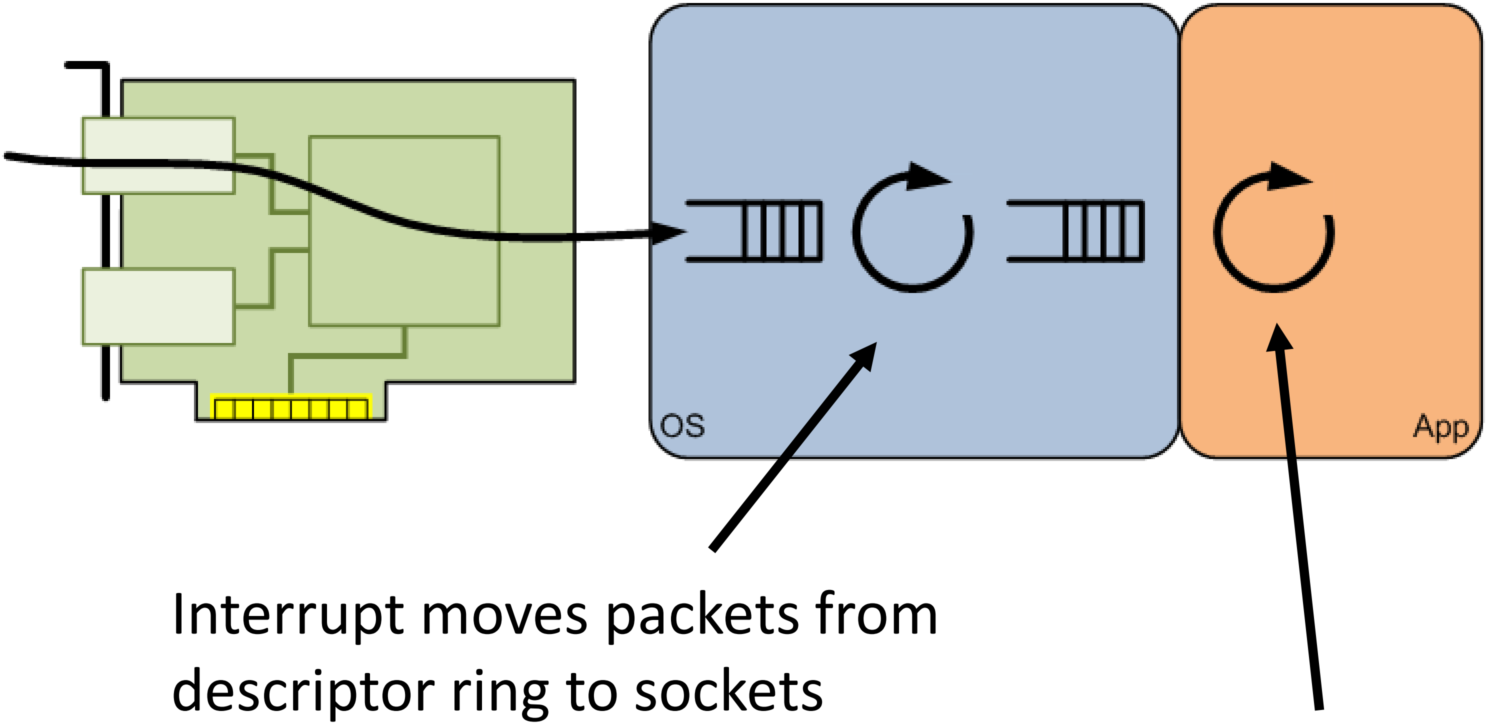
For stable performance when overloaded

Limit working set size

- Limit the sizes of pools, queues, socket buffers etc.

Shed excess load early (Tip 3.1)

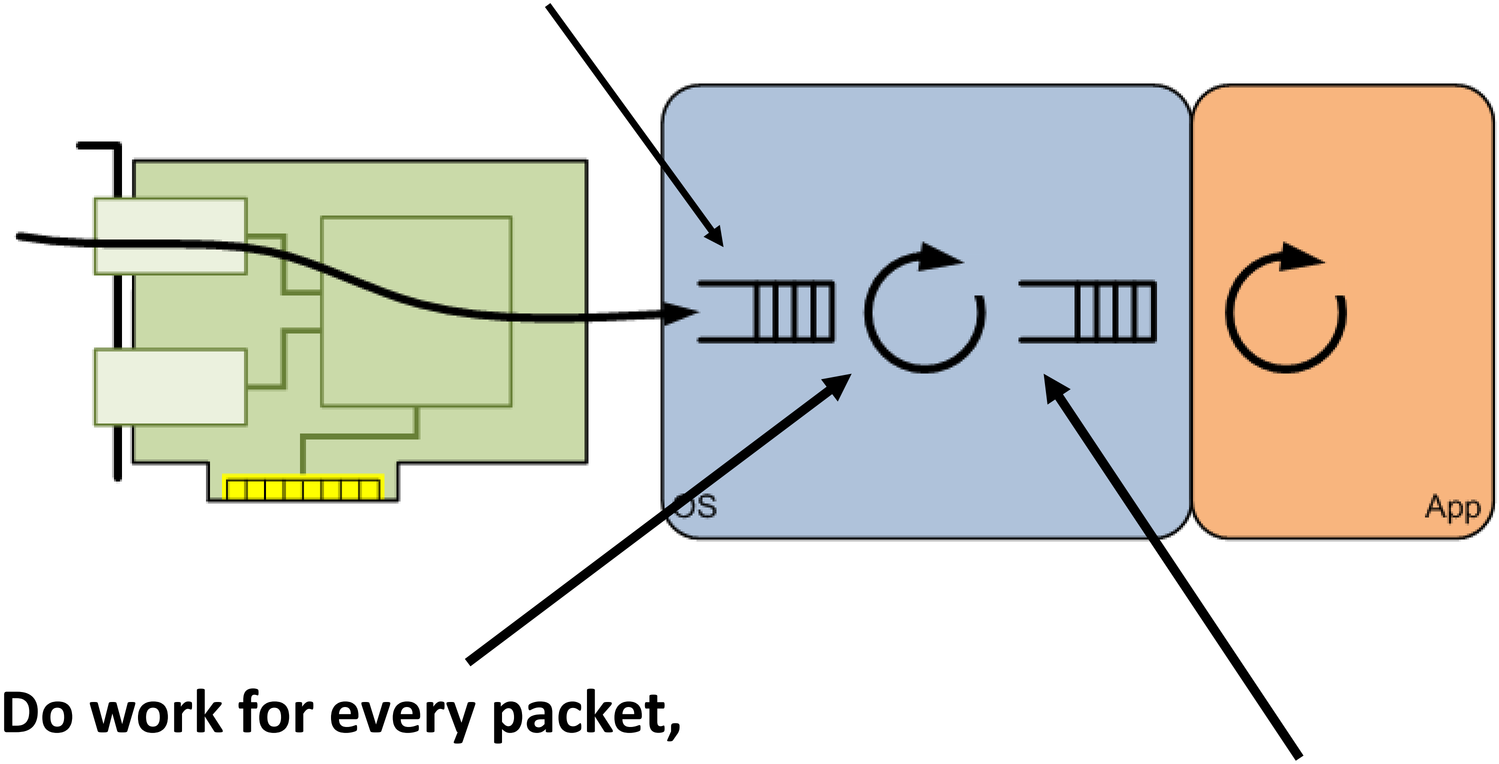
- Before you've wasted time on requests you're going to have to drop anyway



Interrupt moves packets from descriptor ring to sockets

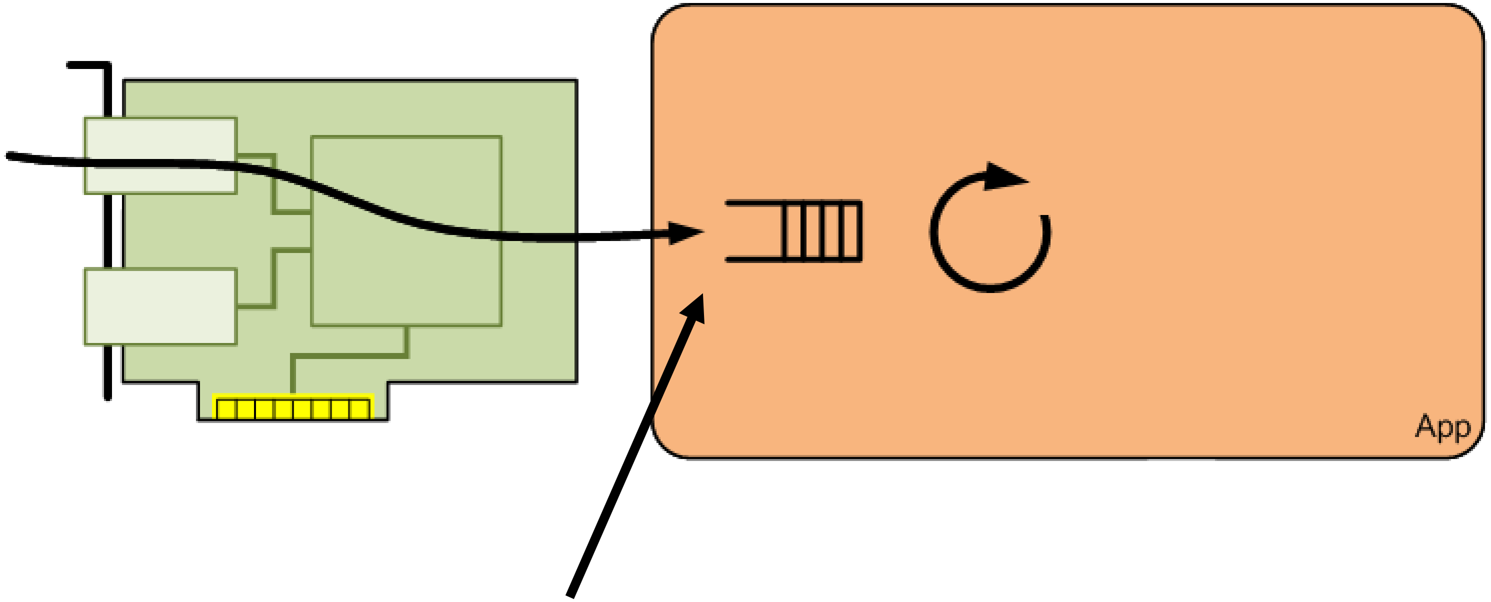
App thread consumes from socket buffer

Drop newest data
(only at very high rates)



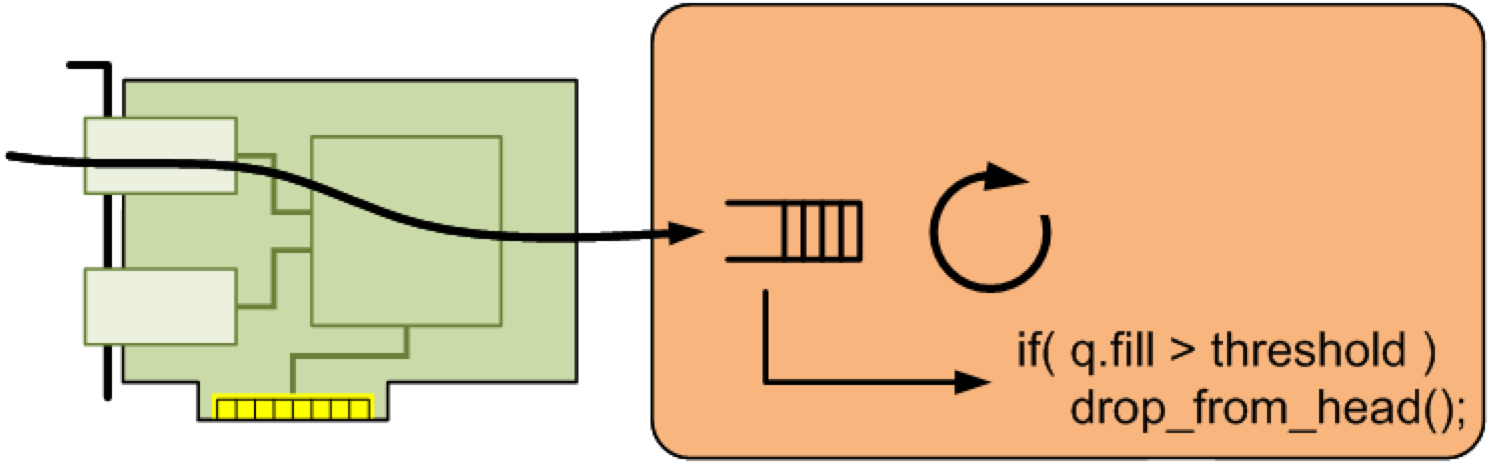
**Do work for every packet,
whether dropped or not**

Drop newest data



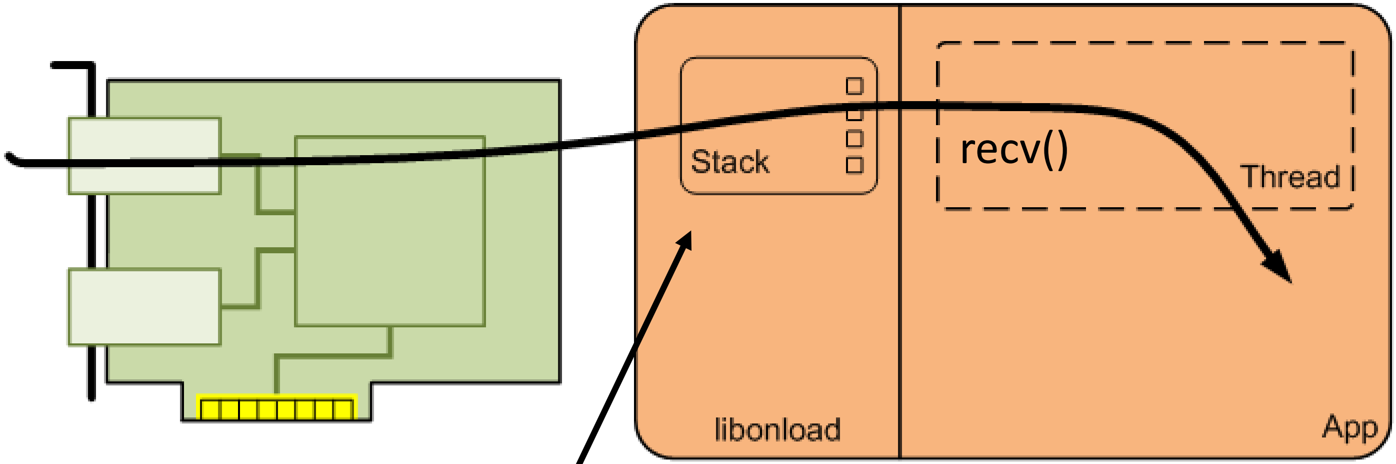
Drop newest data

Tip 3.2 Drop *old* data for better response time

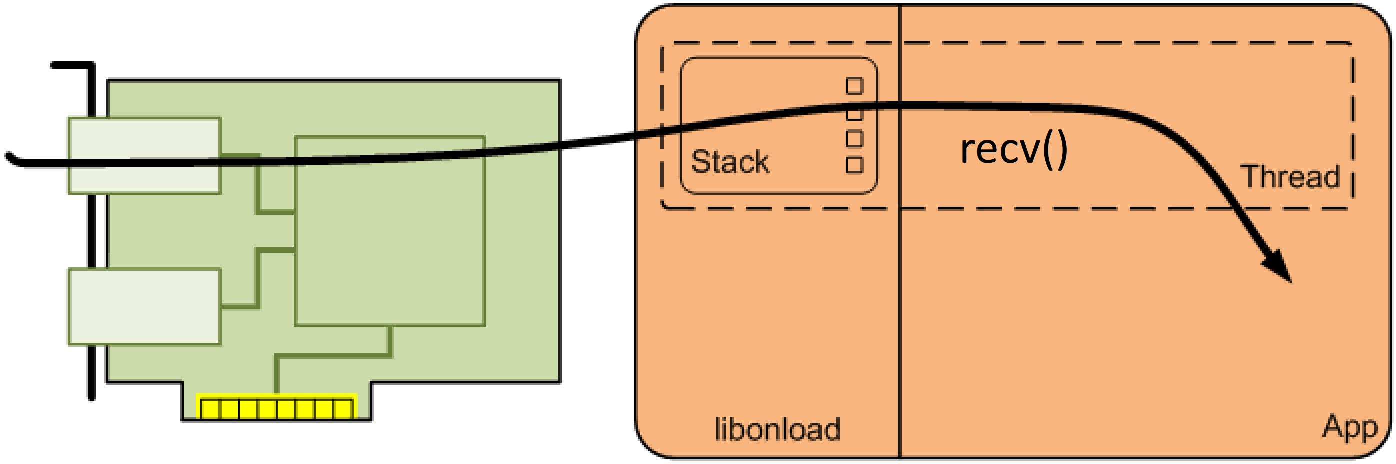


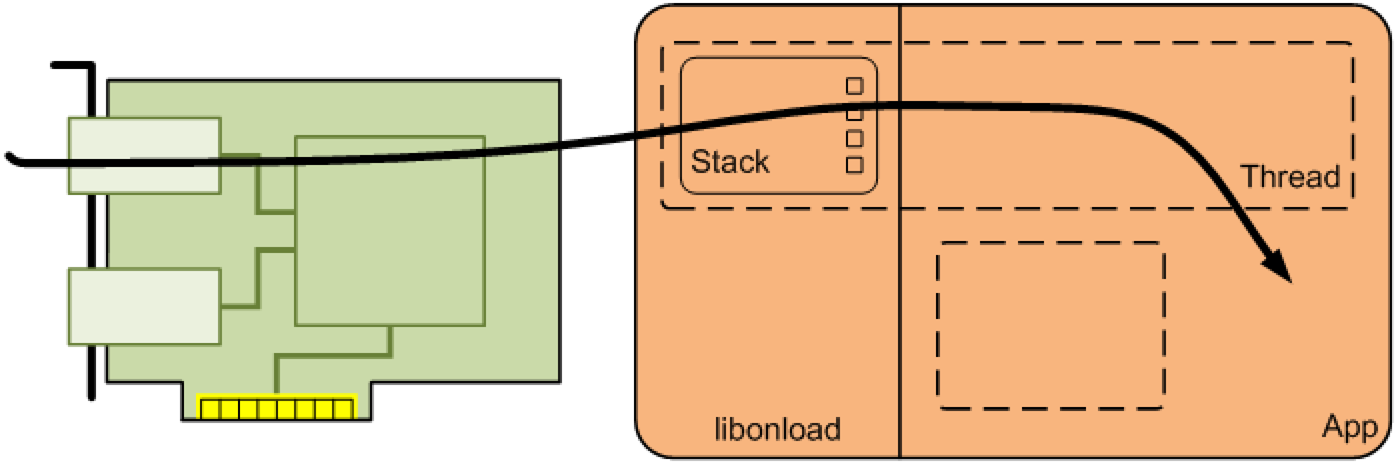
```
if( q.fill > threshold )
    drop_from_head();
```

Last example: Cache locality

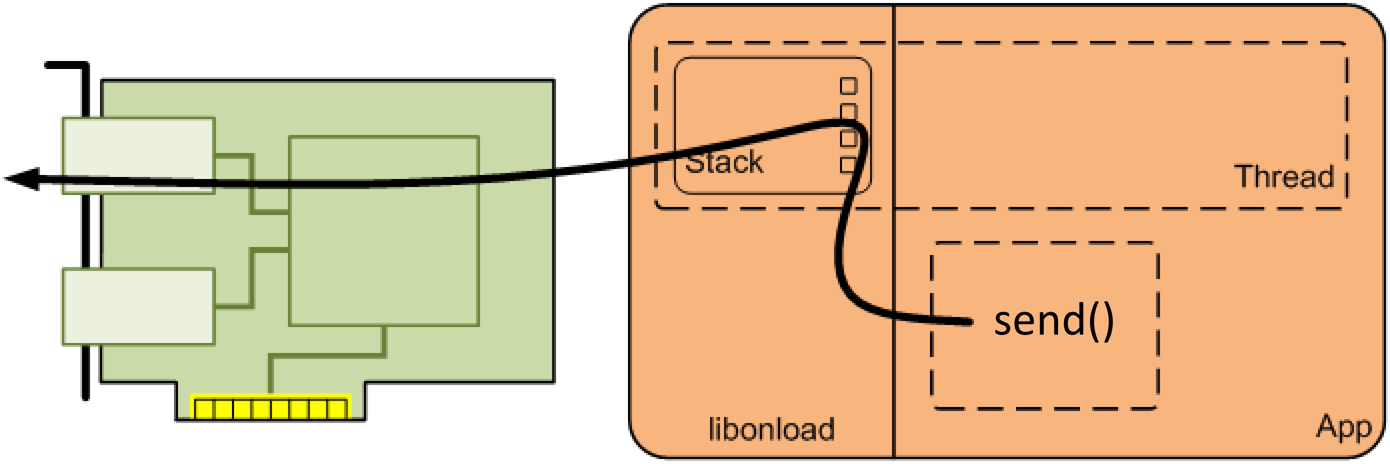


OpenOnload stack; includes sockets, DMA buffers, TX and RX rings, control plane etc.

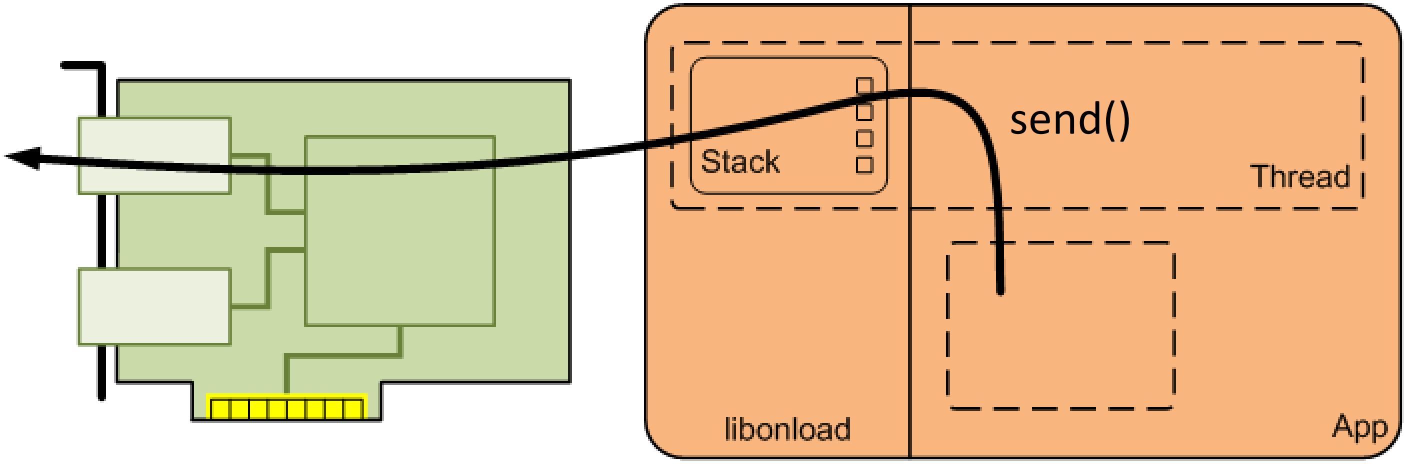

Stack
recv()
Thread
libonload
App



Problem: Send a small (eg. 200 bytes) reply from a different thread

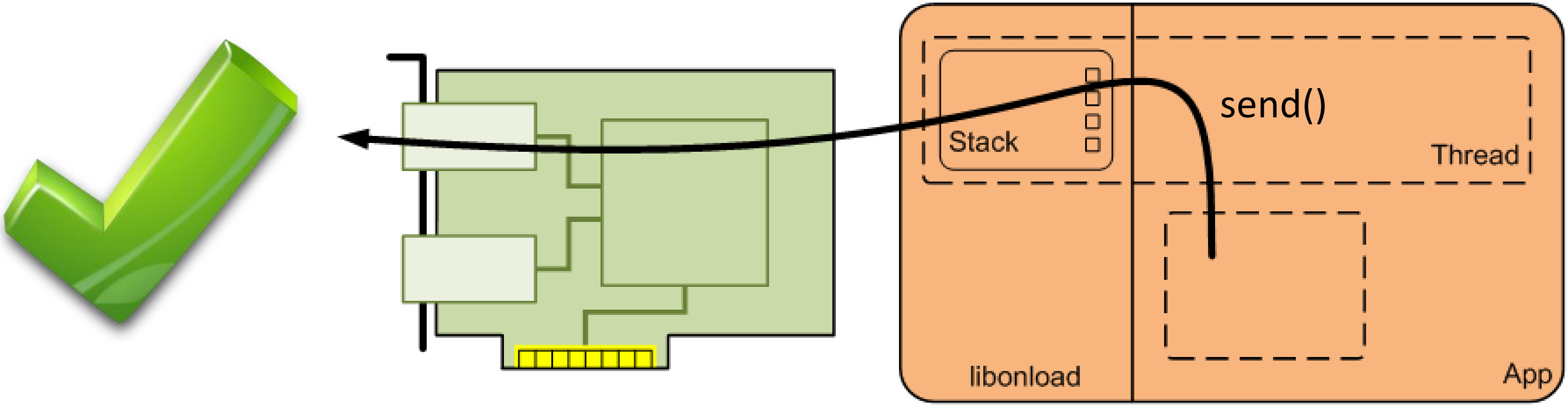

Stack
Thread
send()
libonload
App

Or…


Stack
send()
Thread
libonload
App

- Passing a small message to another thread:
  - A few cache misses

- send() on a socket last accessed on another core:
  - Dozens of cache misses



Thank you!