End-Host Networking

These slides are taken from the lessons of Prof. Gianni Antichi @Polimi

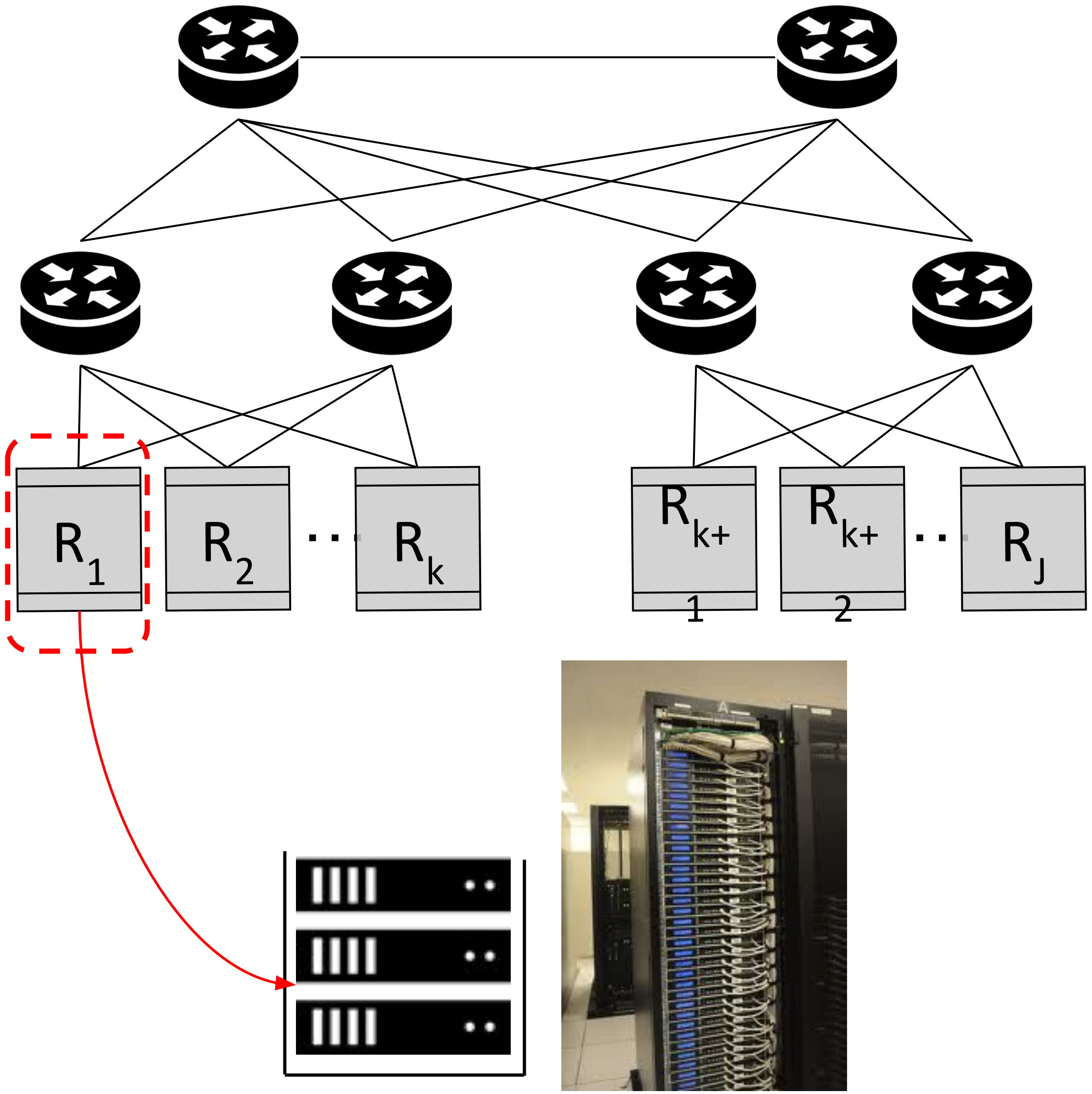
for the Network Computing course

Content

- Life of a packet inside a server
- Kernel Bypass
- Programmable NICs
- Kernel programming (eBPF)

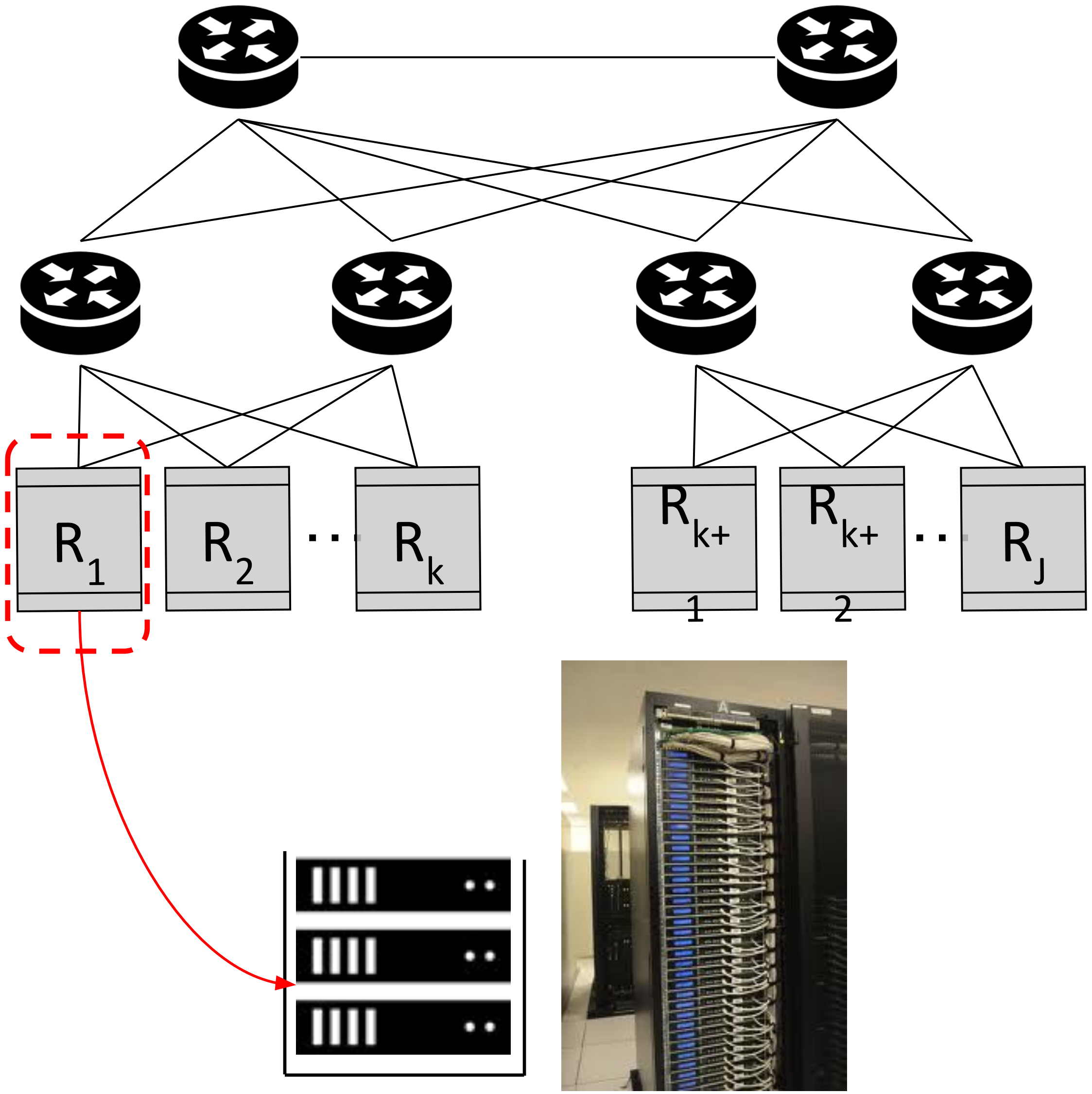
Today’s lecture is about end-host!

- End-hosts (or servers) are located in racks
- This is where applications run and traffic is generated/received



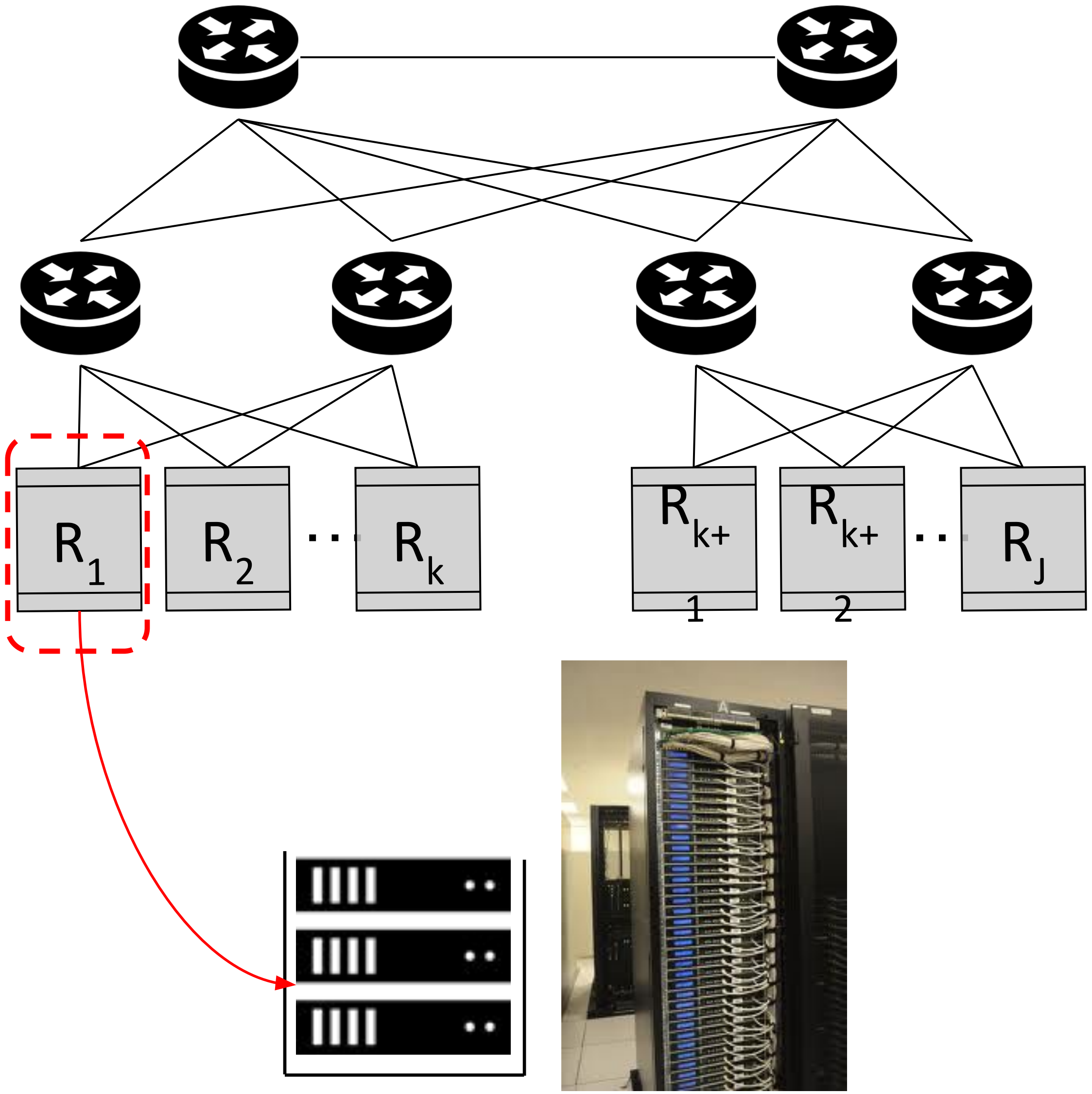
Today’s lecture is about end-host!

- Lots of efforts in improving end-host stacks
- Deploy *#fancy_new_hardware* in-network is not always the best idea (from an operational standpoint)
    - New hardware == new problems
    - Deployments take time
    - Financial costs
    - Cabling
    - Many applications: who do you deploy the hardware for?



Today is about end-host!

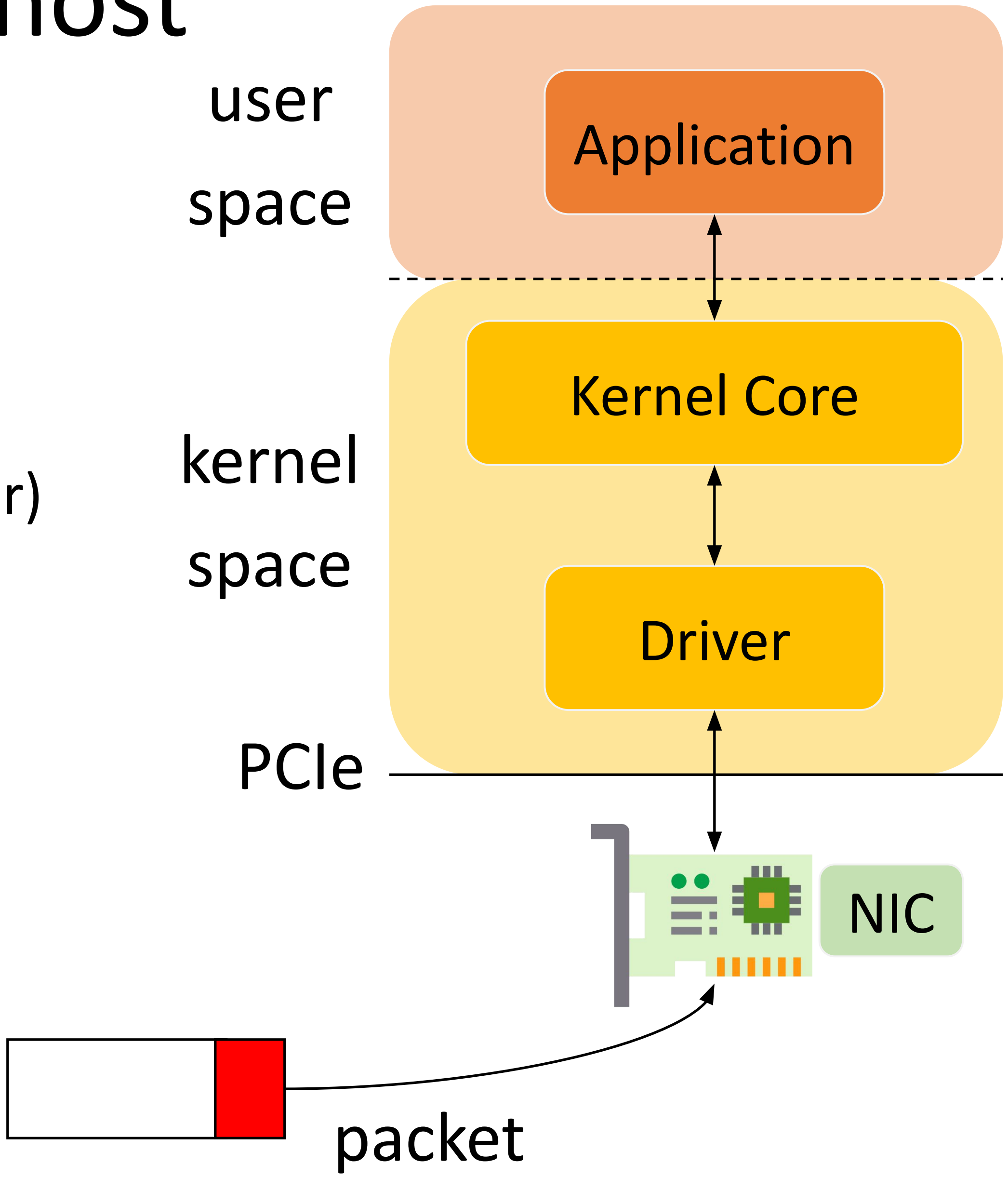
- Deploy *#fancy_new_funcionality* at the end-host is easier
  - You are also closer to where applications are running
  - Less disruption on the network
  - More possibility to test new solutions



Life of a packet at the end-host

- From an high level perspective every packet has to cross:
  1. Network Interface Card (NIC)
  2. PCIe (interconnect between NIC and server)
  3. NIC driver
  4. Kernel

- …to finally reach the application in user space



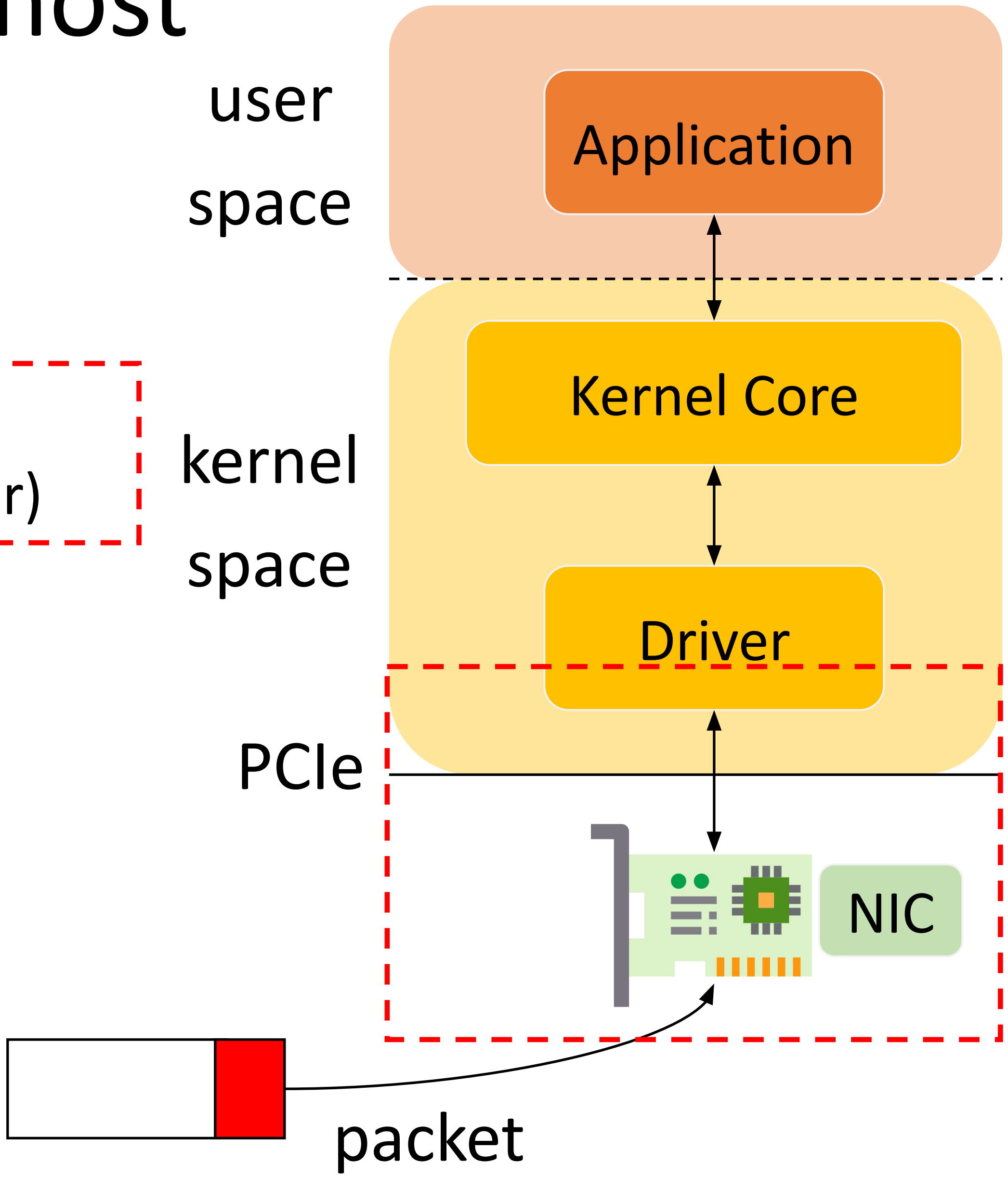
The hardware side

The NIC and the PCIe

Life of a packet at the end-host

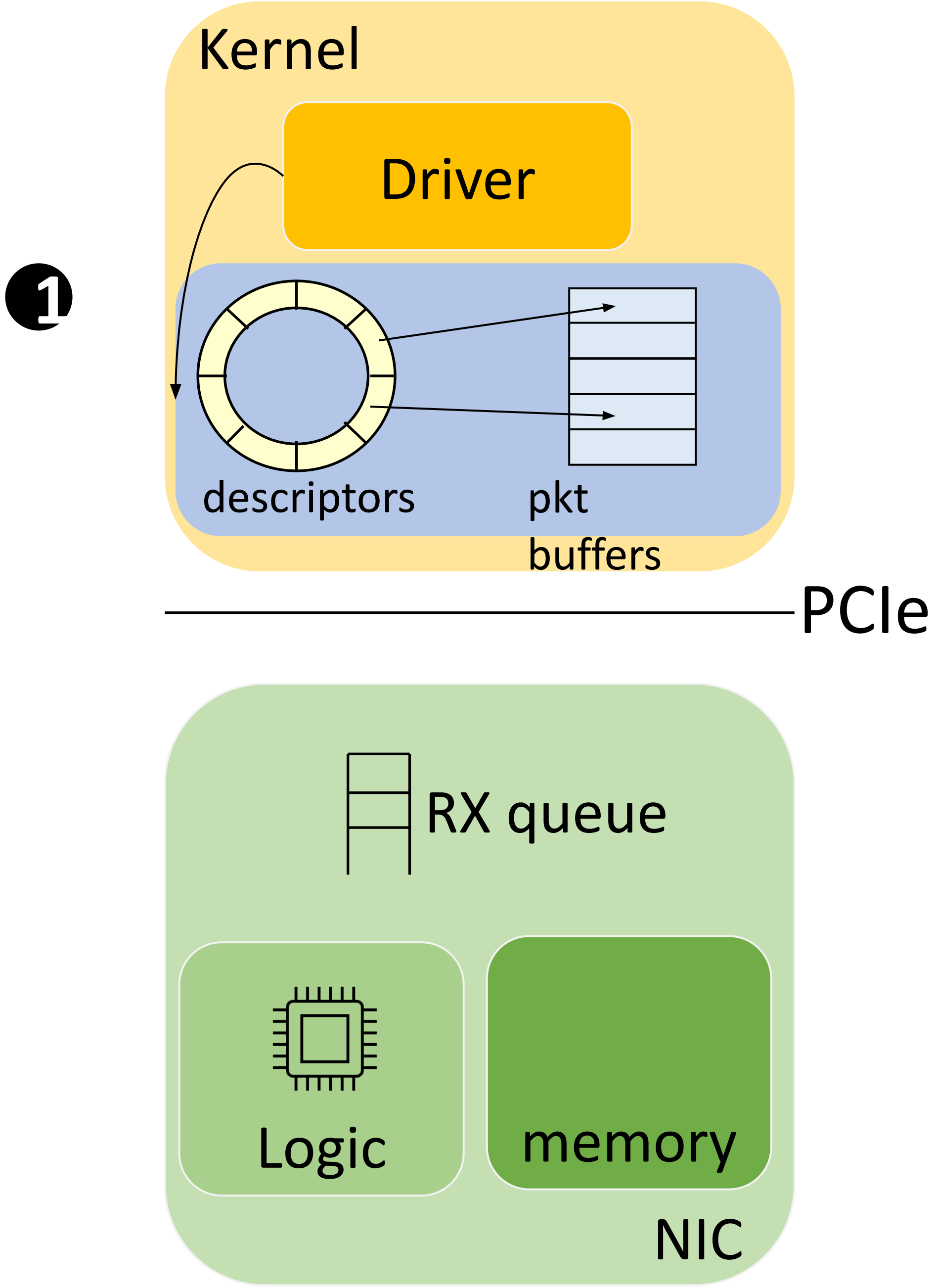
- From an high level perspective every packet has to cross:
  1. Network Interface Card (NIC)
  2. PCIe (interconnect between NIC and server)
  3. NIC driver
  4. Kernel

- …to finally reach the application in user space



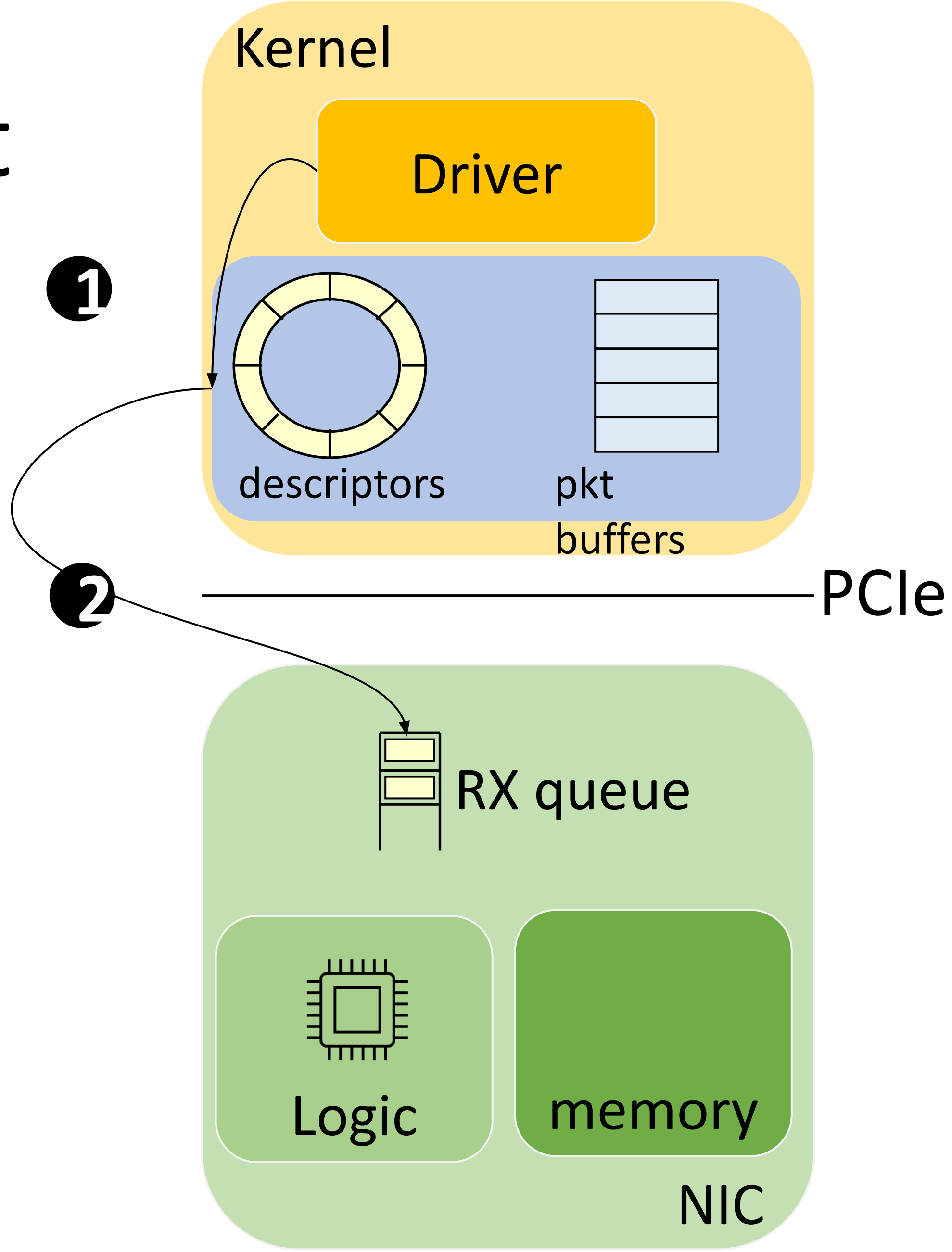
Life of a packet at the end-host

- The kernel allocates memory for storing the received packets from the NIC (*pkt buffers*)
- A descriptors ring is kept to store pointers to those buffers



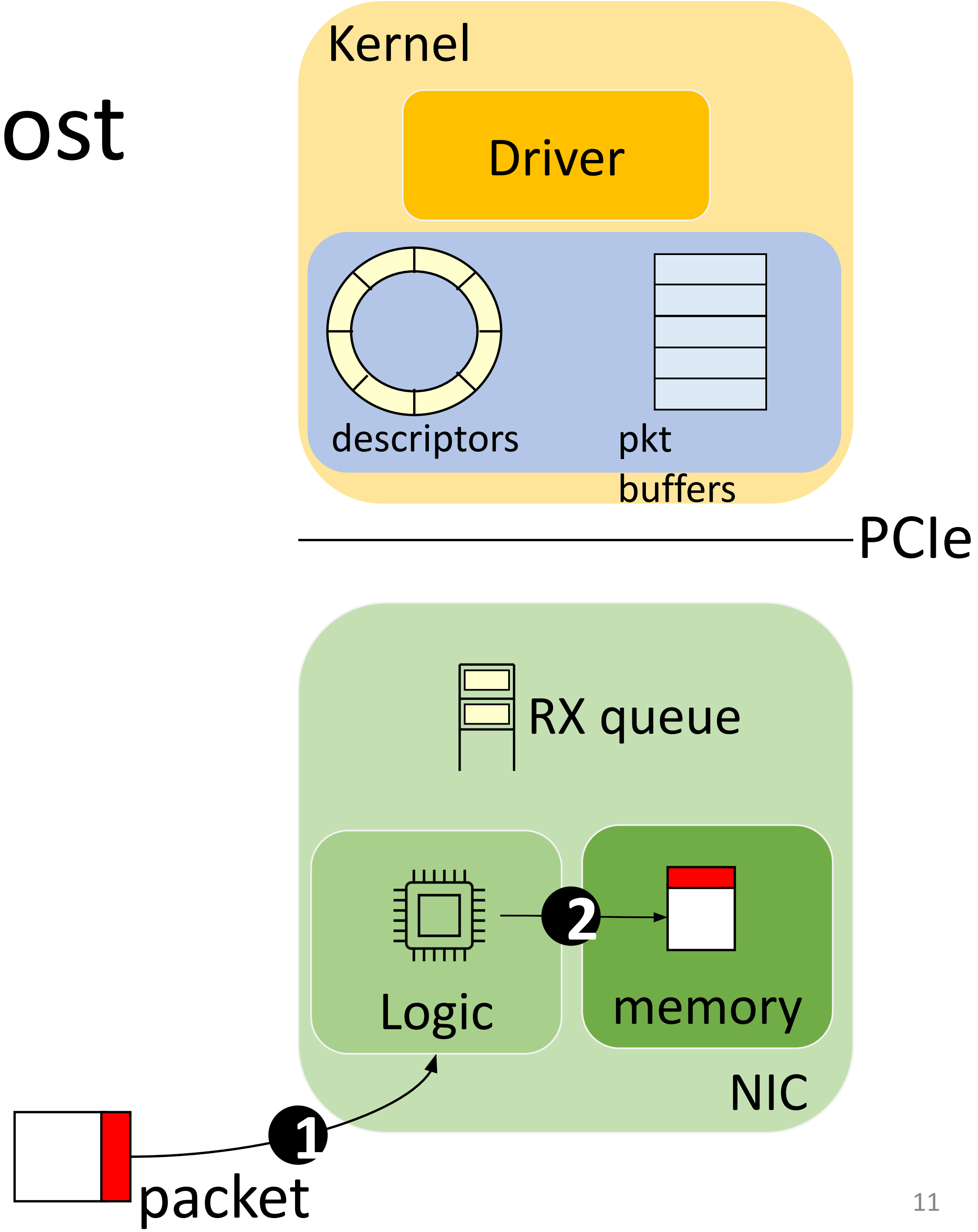
Life of a packet at the end-host

- The kernel allocates memory for storing the received packets from the NIC (*pkt buffers*)
- A descriptors ring is kept to store pointers to those buffers
- The driver populates the RX queue in the NIC with available descriptors (i.e., associated pkt buffers are empty)



Life of a packet at the end-host

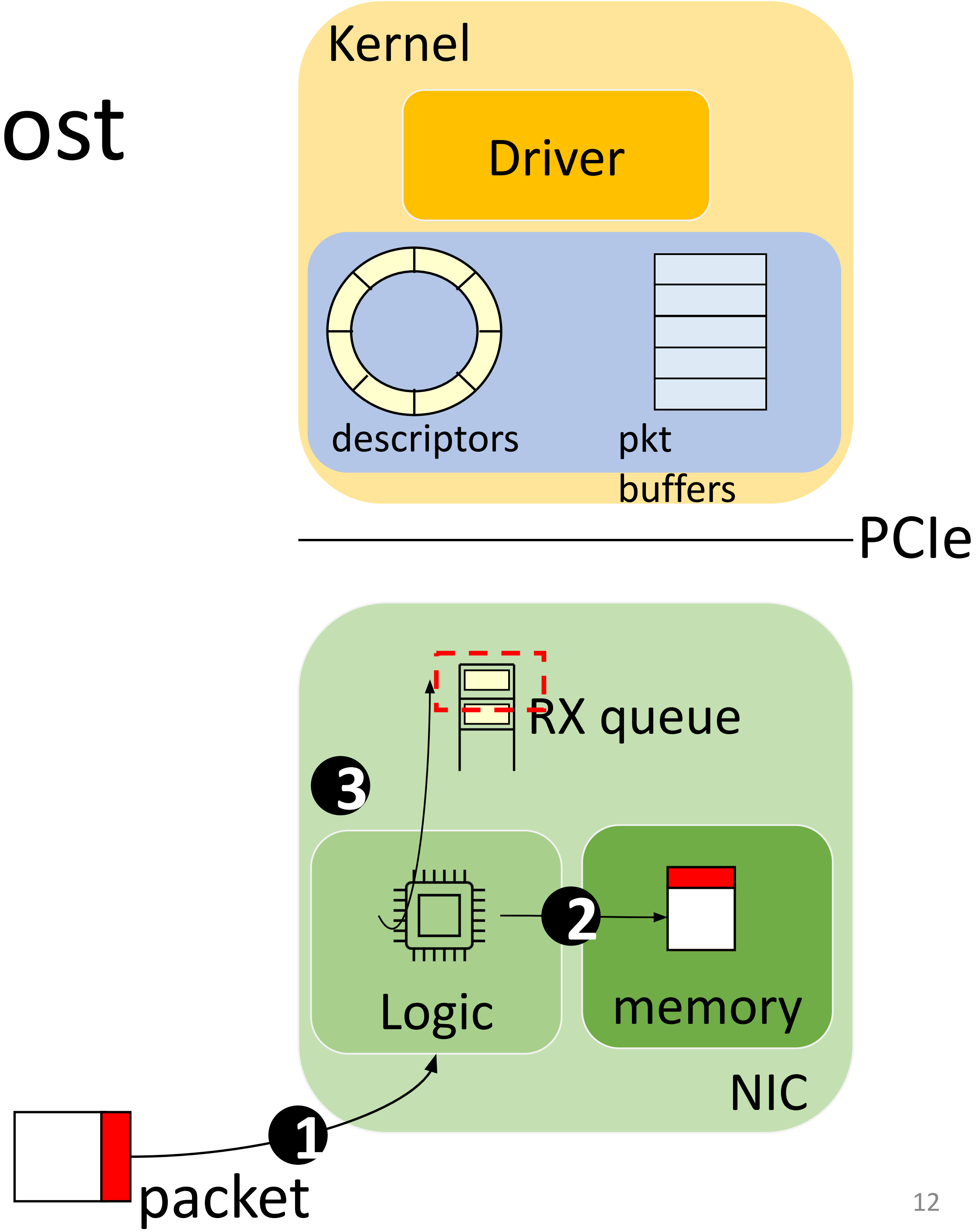
- When the packet arrives at the NIC, it is first stored in its local memory



Life of a packet at the end-host

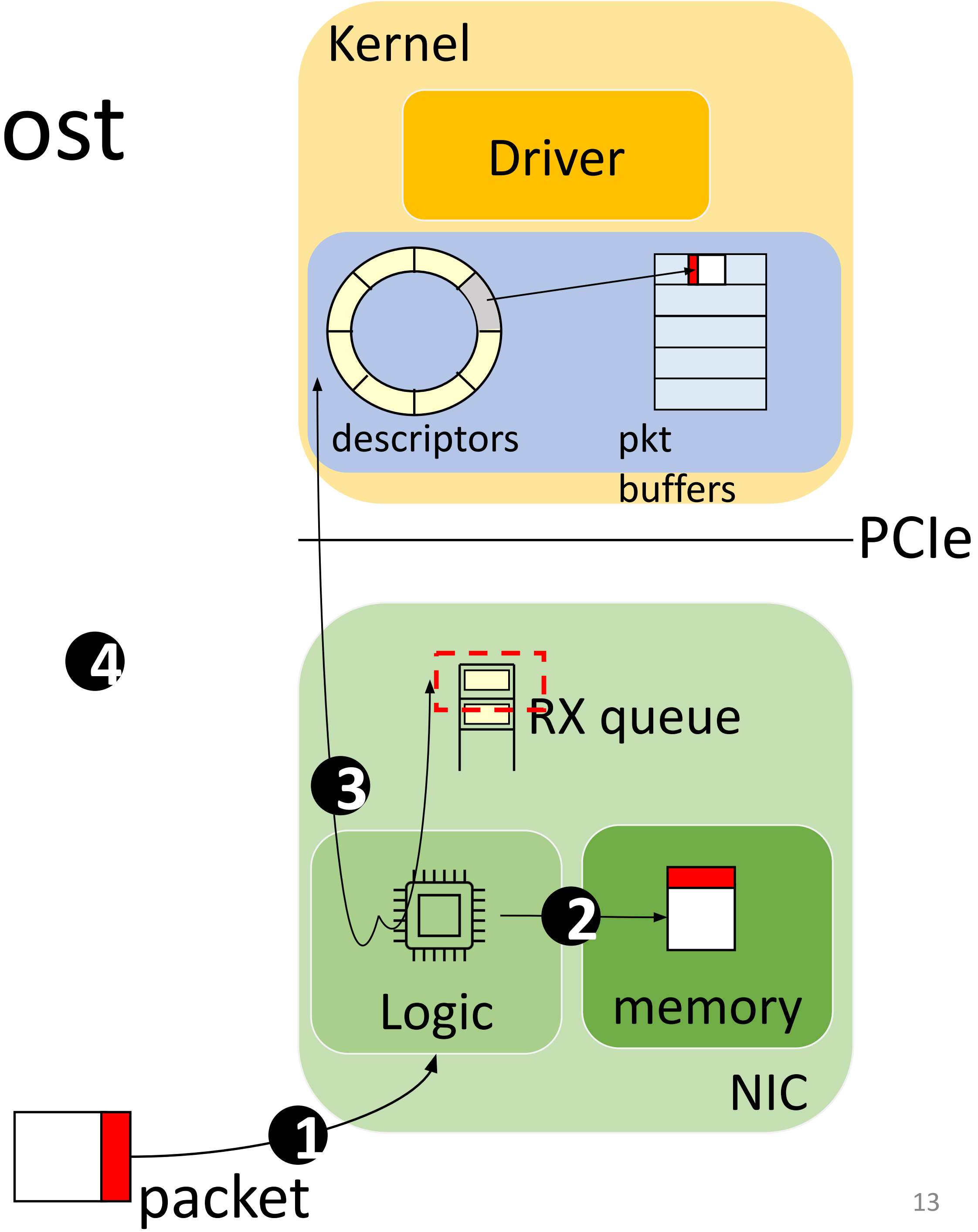
- When the packet arrives at the NIC, it is first stored in its local memory

- Then the NIC fetches one descriptor from the RX queue so it will know where to transfer the packet in the host memory



Life of a packet at the end-host

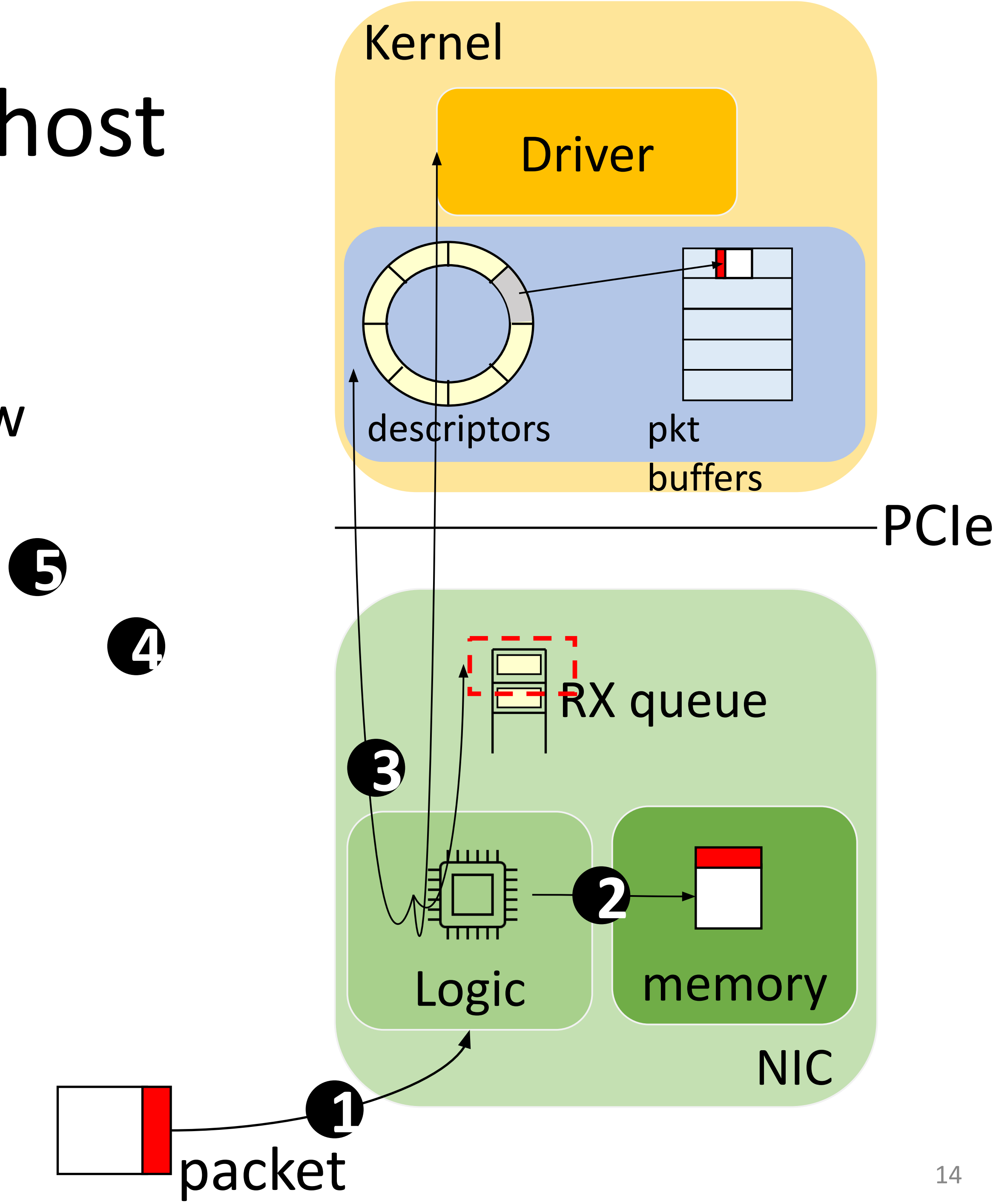
- When the packet arrives at the NIC, it is first stored in its local memory
- Then the NIC fetches one descriptor from the RX queue so it will know where to transfer the packet in the host memory
- The NIC starts a Direct Memory Access (DMA) transaction over PCIe to move the packet from NIC to host memory (*NO CPU involvement*) and update associated descriptor



Life of a packet at the end-host

- Finally the NIC generates an Interrupt ReQuest (IRQ) to inform the driver of new data to be processed

- A CPU core will take the IRQ and the processing on the host starts

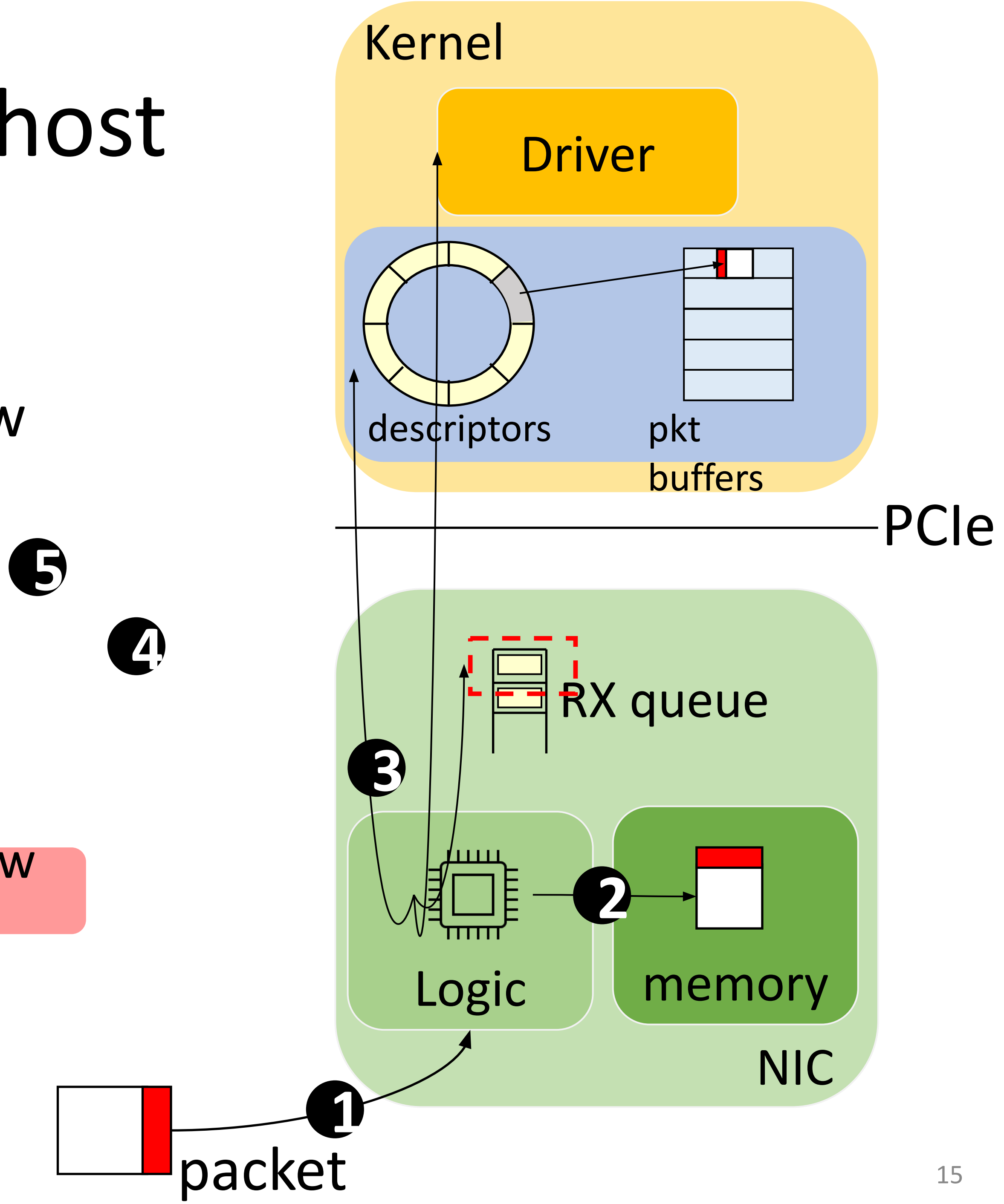


Life of a packet at the end-host

- Finally the NIC generates an Interrupt ReQuest (IRQ) to inform the driver of new data to be processed

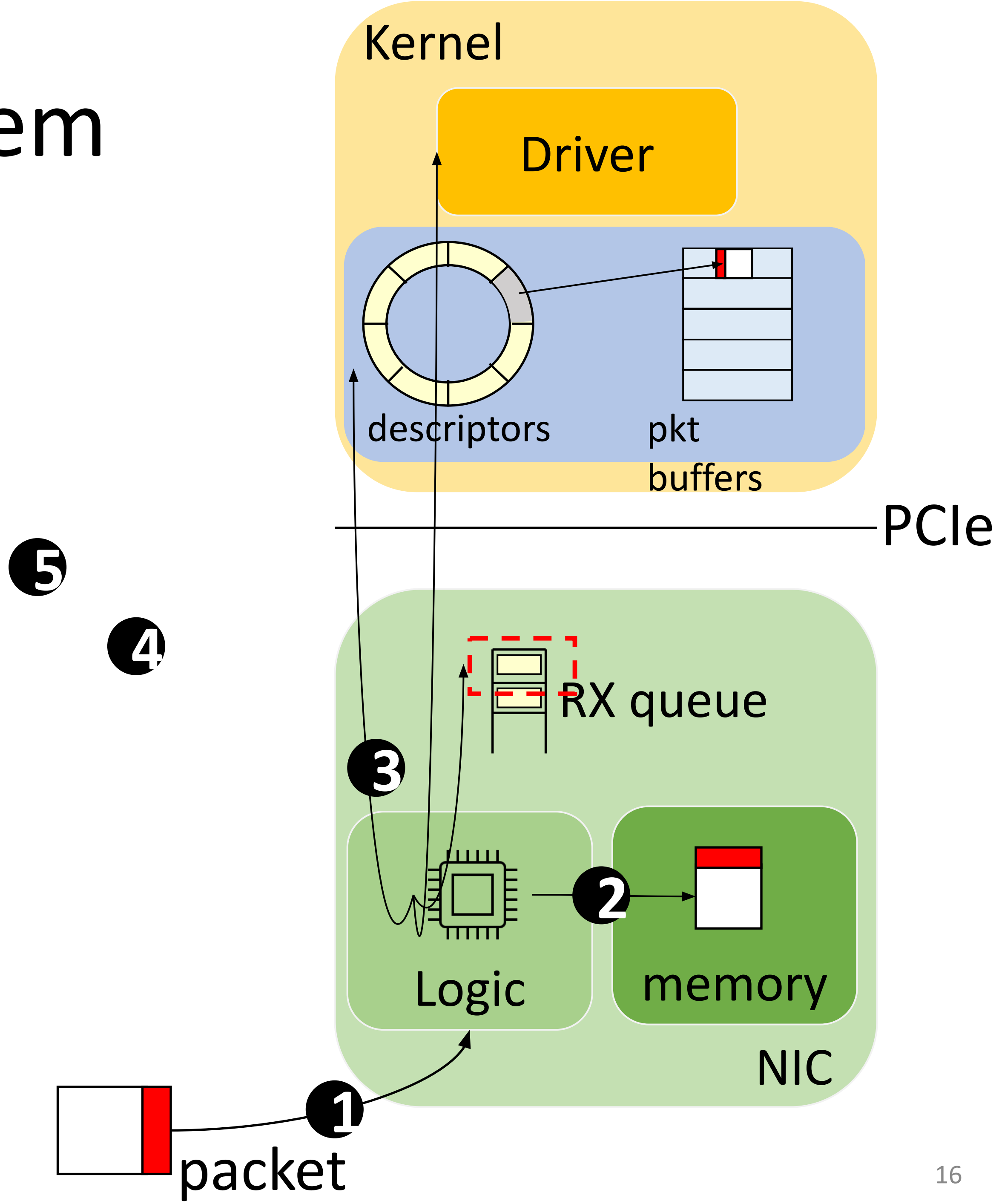
- A CPU core will take the IRQ and the processing on the host starts

This is a very simplified view ☺



The Receive Livelock problem

- Per-packet interrupts can cause performance degradation at high-rates
- Receiver interrupts take priority over all other activities. If packets arrive too fast, the system will spend all of its time processing receiver interrupts

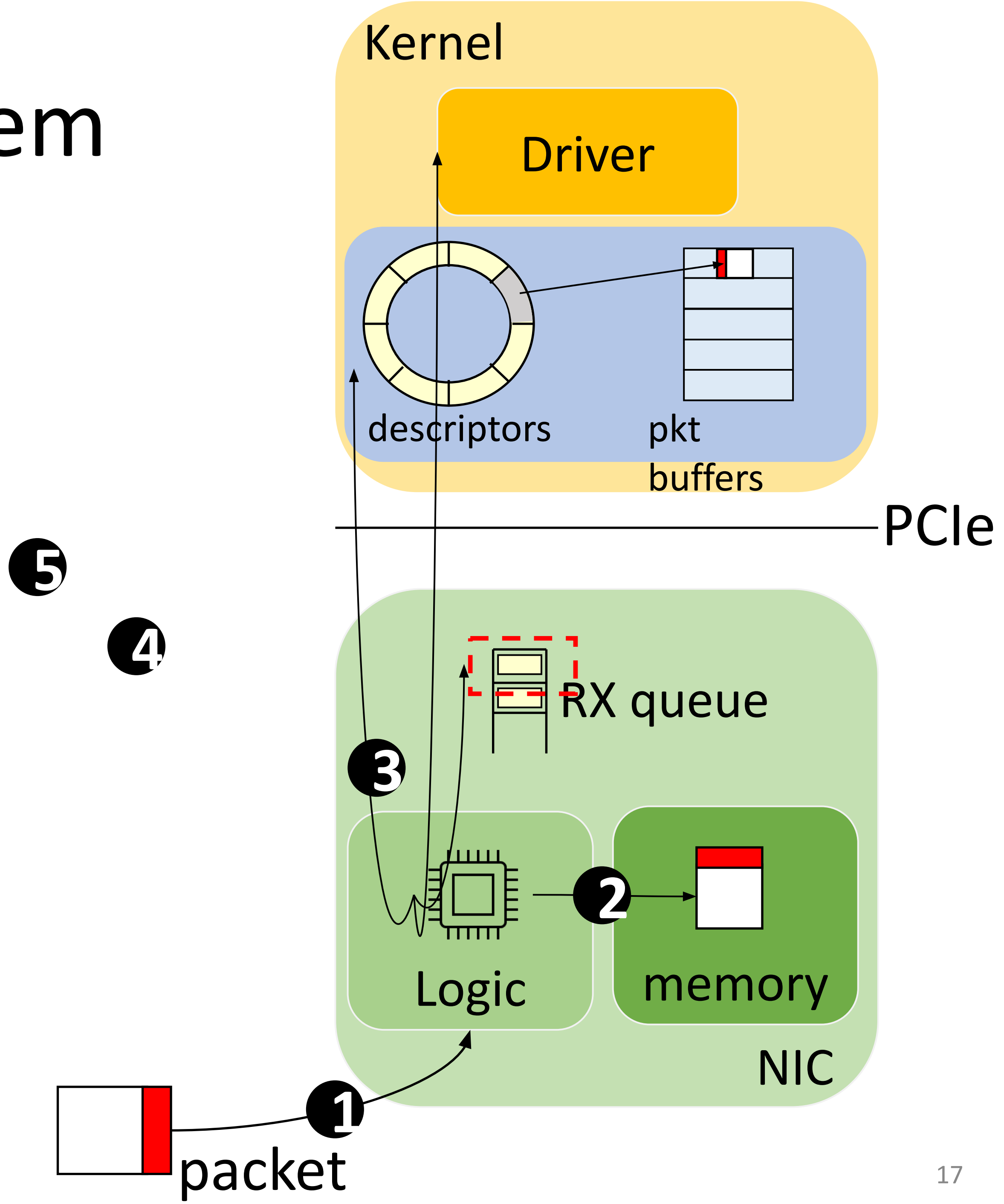


The Receive Livelock problem

- Per-packet interrupts can cause performance degradation at high-rates

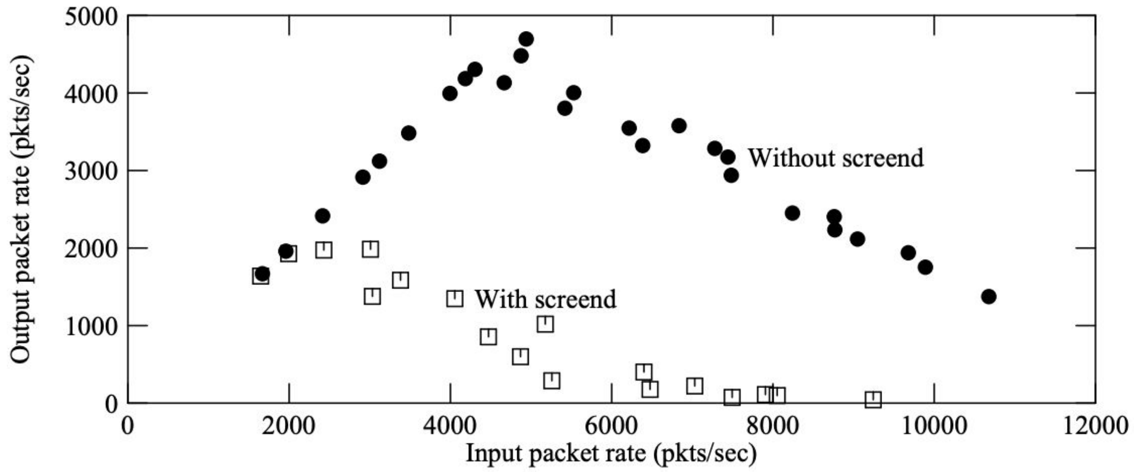
- Receiver interrupts take priority over all other activities. If packets arrive too fast, the system will spend all of its time processing receiver interrupts

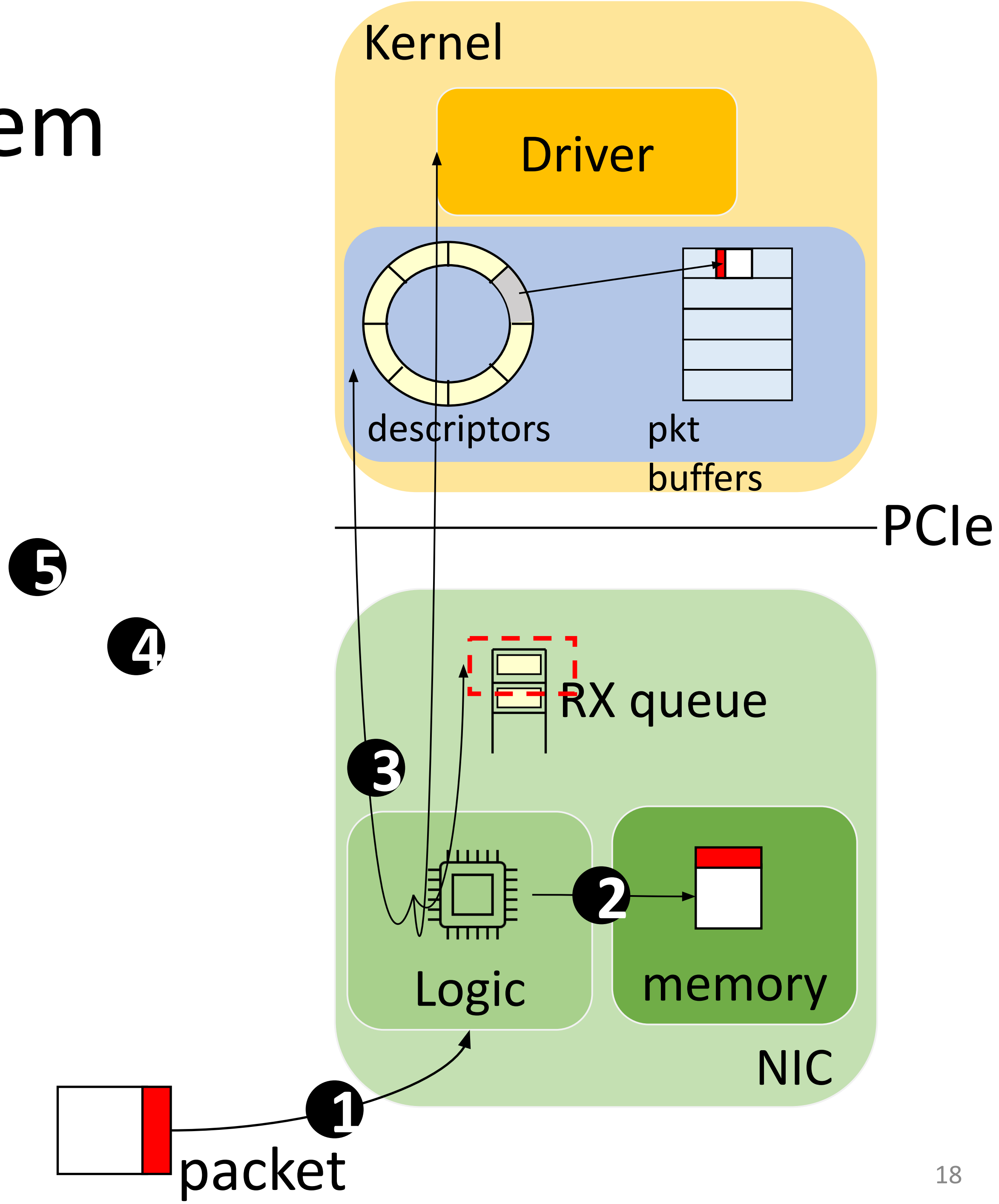
Consequence: no resources left for processing and the system throughput will drop to zero



The Receive Livelock problem

kernel-based forwarding on a 10Mbps ethernet

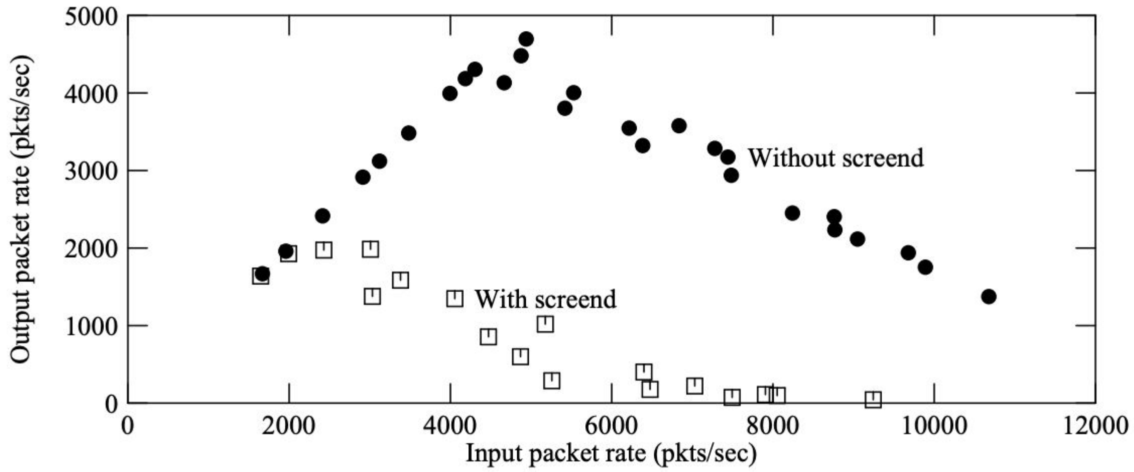




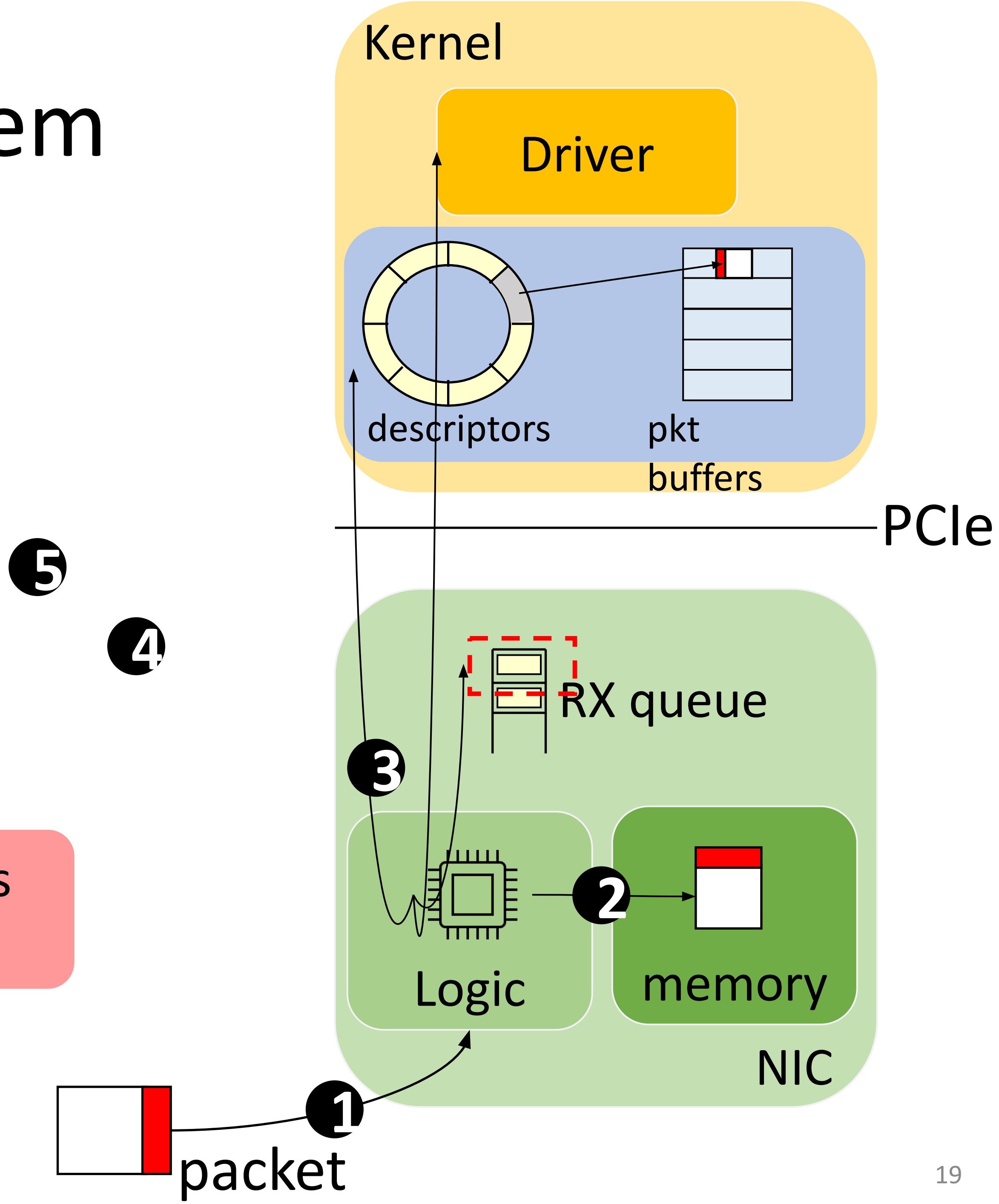
Eliminating Receive Livelock in an Interrupt-driven Kernel (USENIX NSDI)

The Receive Livelock problem

kernel-based forwarding on a 10Mbps ethernet



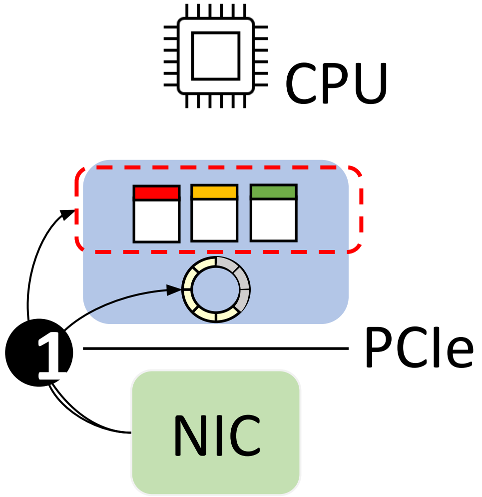
This is an old system (1996), but same concepts apply to newer CPUs as line rate increases



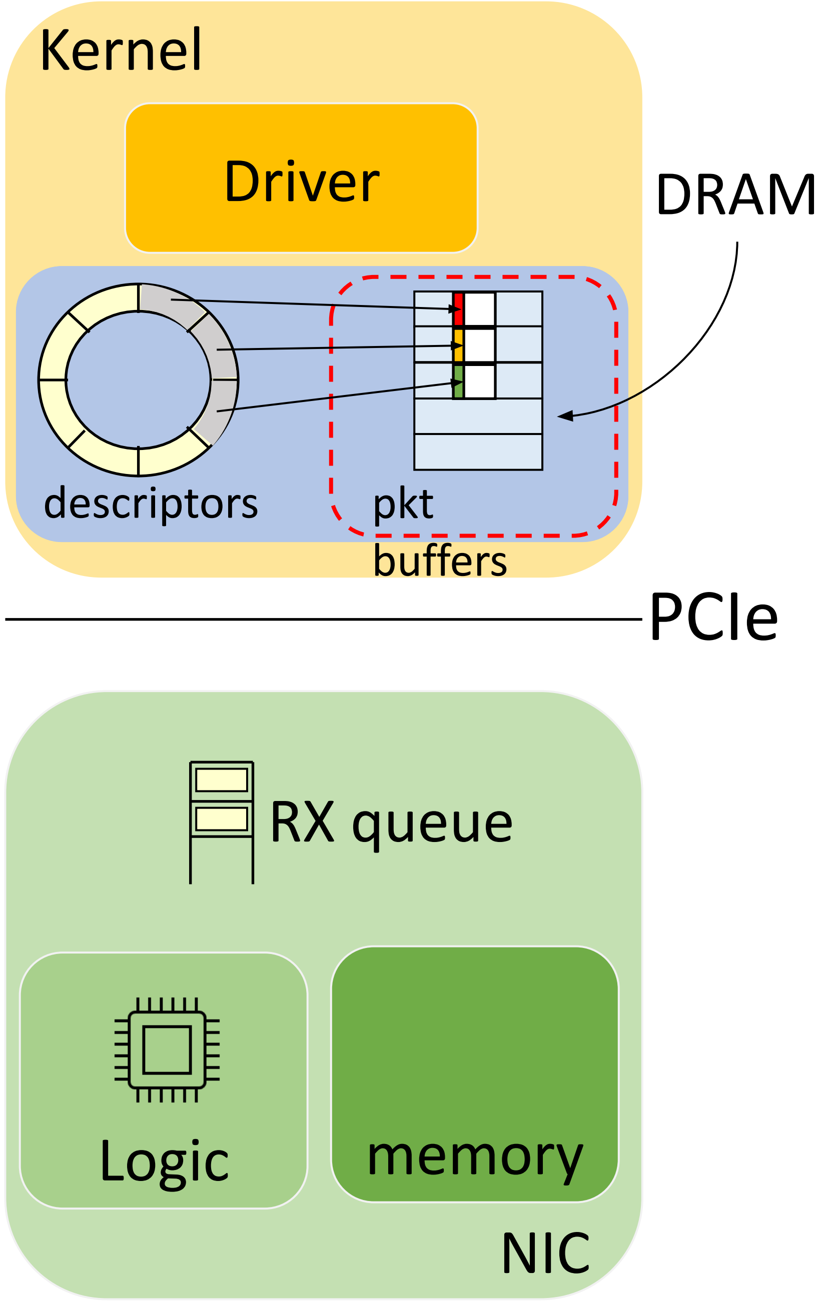
Eliminating Receive Livelock in an Interrupt-driven Kernel (USENIX NSDI)

Interrupt mitigation strategies

- Modern NICs provide ways to mitigate the number of interrupts sent from HW to the CPU
  - Interrupt coalescing delay interrupts and process multiple events at once with a compromise between reaction time and overhead



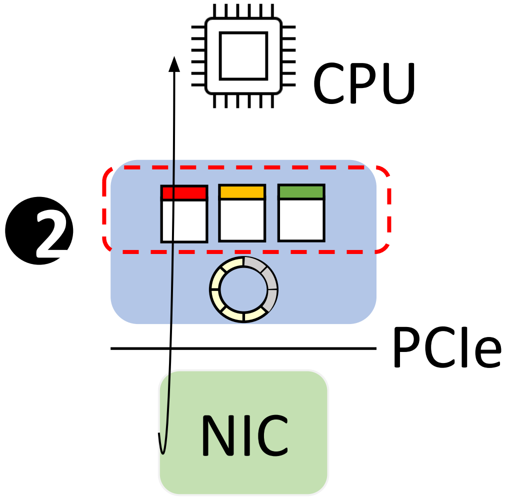
NIC stores a batch of packets in DRAM and updates descriptors ring

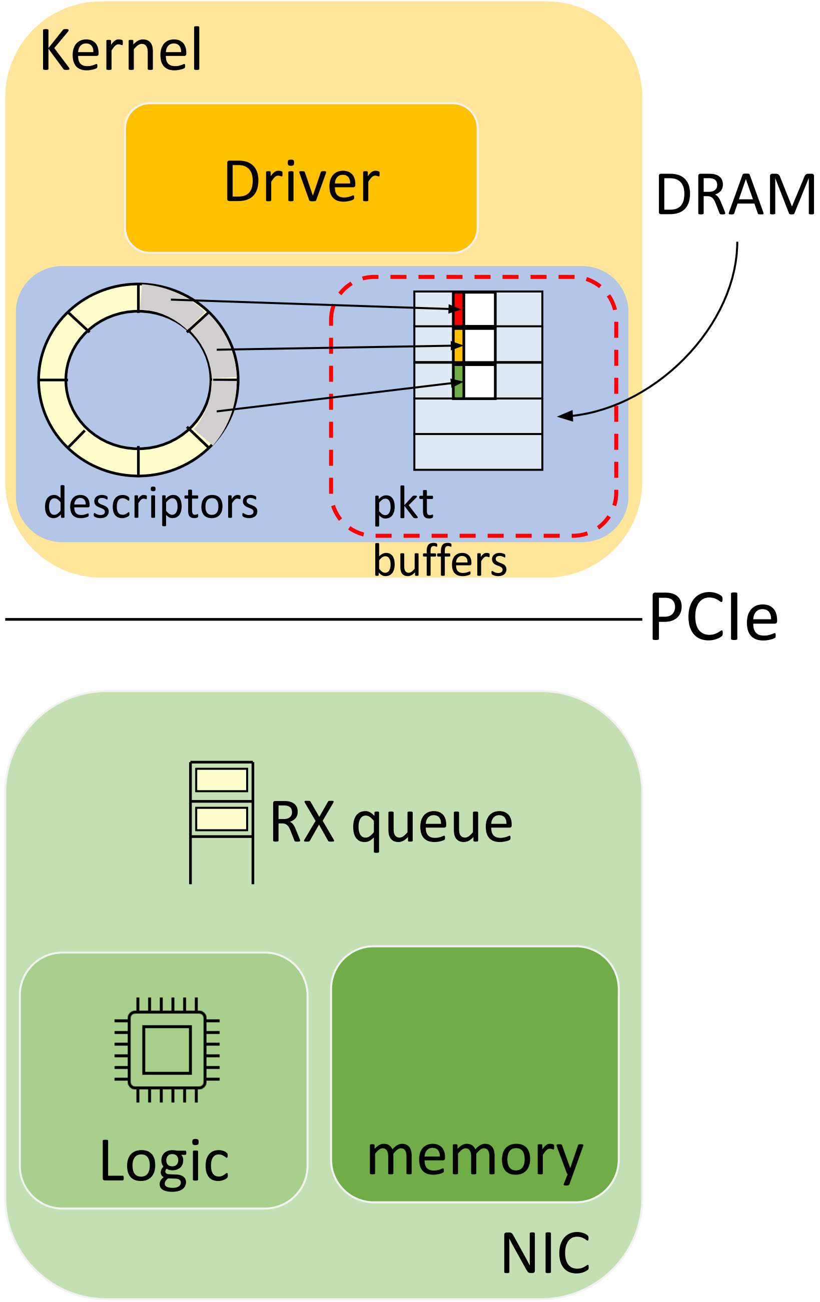


Interrupt mitigation strategies

- Modern NICs provide ways to mitigate the number of interrupts sent from HW to the CPU
  - Interrupt coalescing delay interrupts and process multiple events at once with a compromise between reaction time and overhead

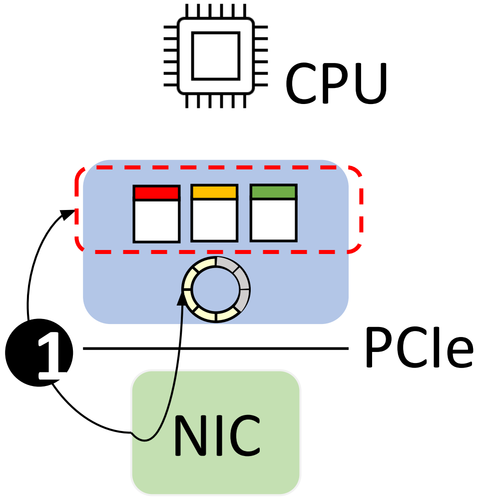
NIC triggers one interrupt valid for the whole batch



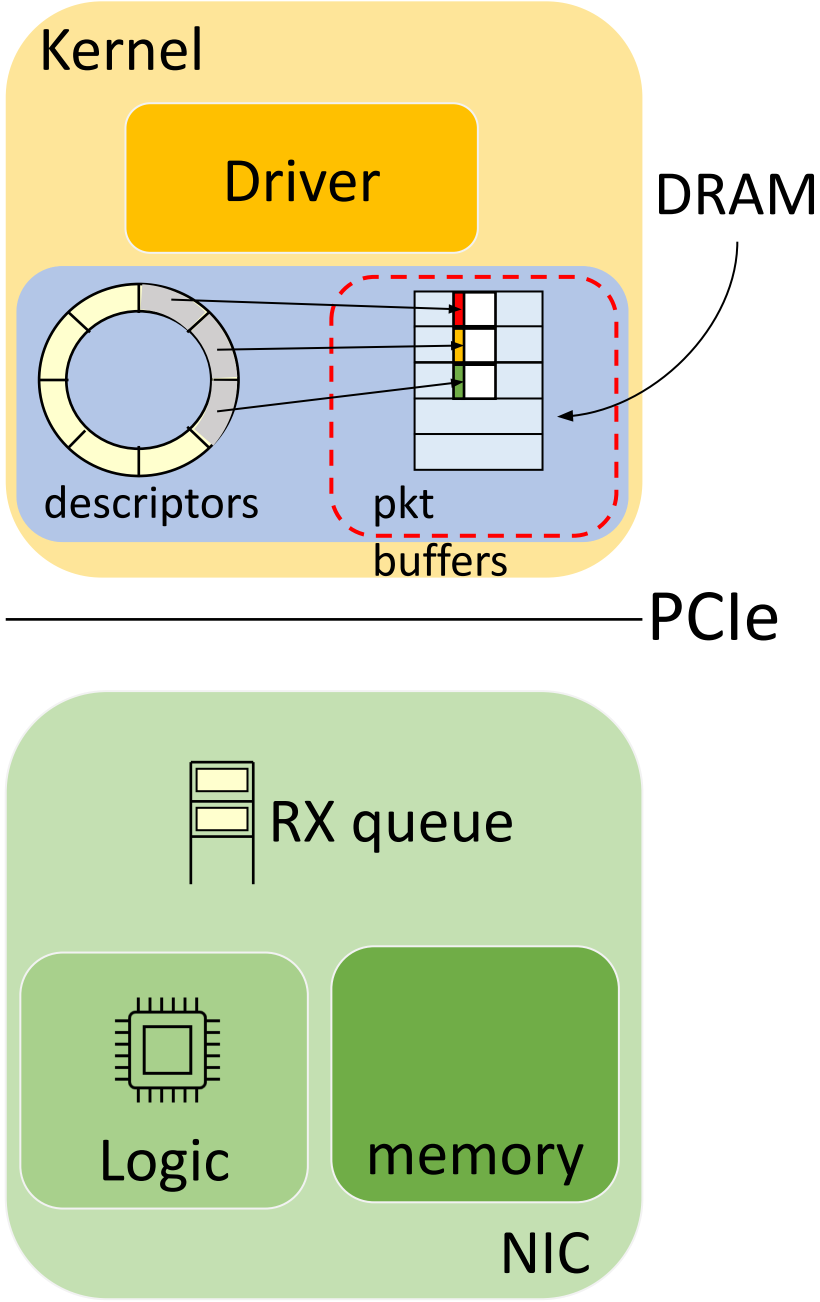


Interrupt mitigation strategies

- Modern NICs provide ways to mitigate the number of interrupts sent from HW to the CPU
  - Polling the CPU continuously check if the device has anything to send. It relies on busy-polling loop potentially wasting resources.



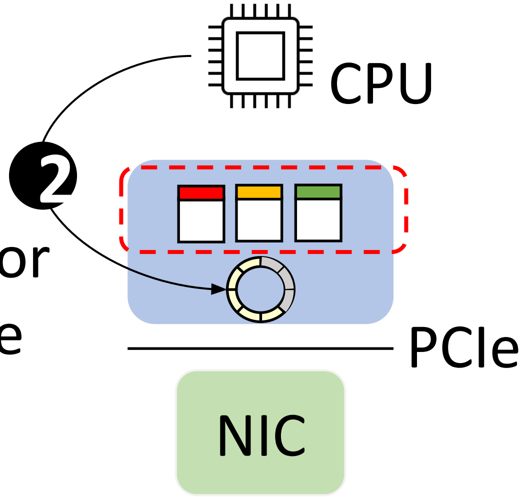
NIC stores packets in DRAM and updates descriptors ring

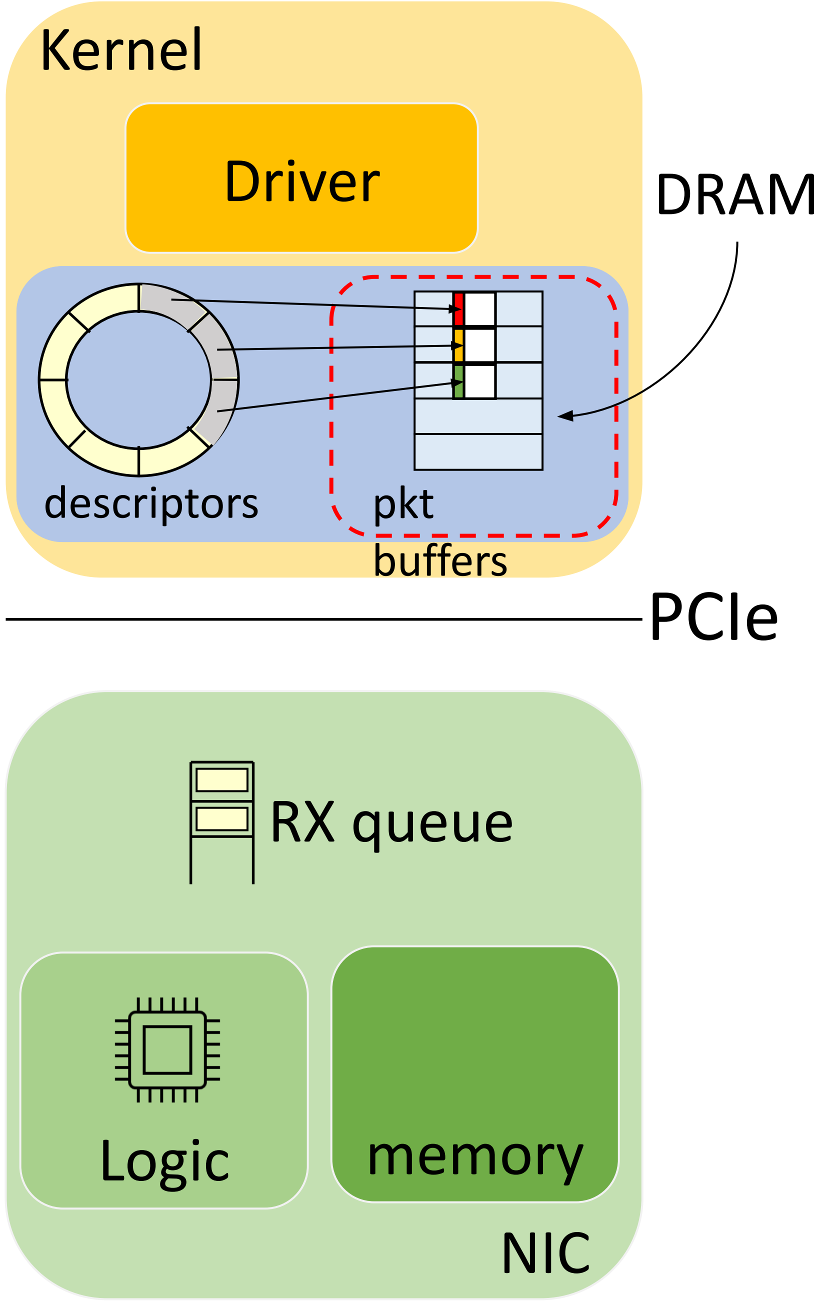


Interrupt mitigation strategies

- Modern NICs provide ways to mitigate the number of interrupts sent from HW to the CPU
  - Polling the CPU continuously check if the device has anything to send. It relies on busy-polling loop potentially wasting resources.

The CPU keeps spinning looking for new packets in the descriptors ring

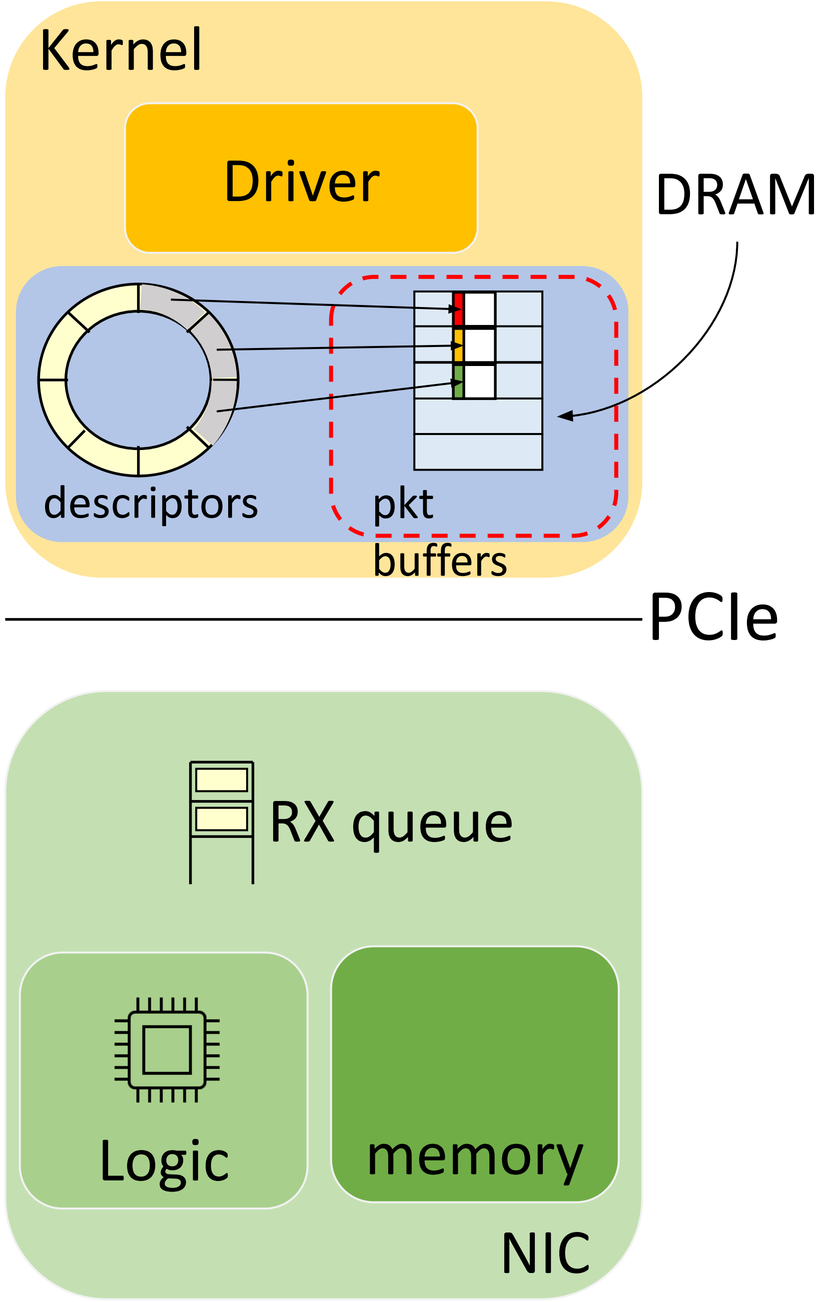




Interrupt mitigation strategies

- Modern NICs provide ways to mitigate the number of interrupts sent from HW to the CPU
  - NAPI adaptively switching between interrupt and polling according to the current workload
  - Low load: interrupt-based
  - High load: CPU-polling

Why?

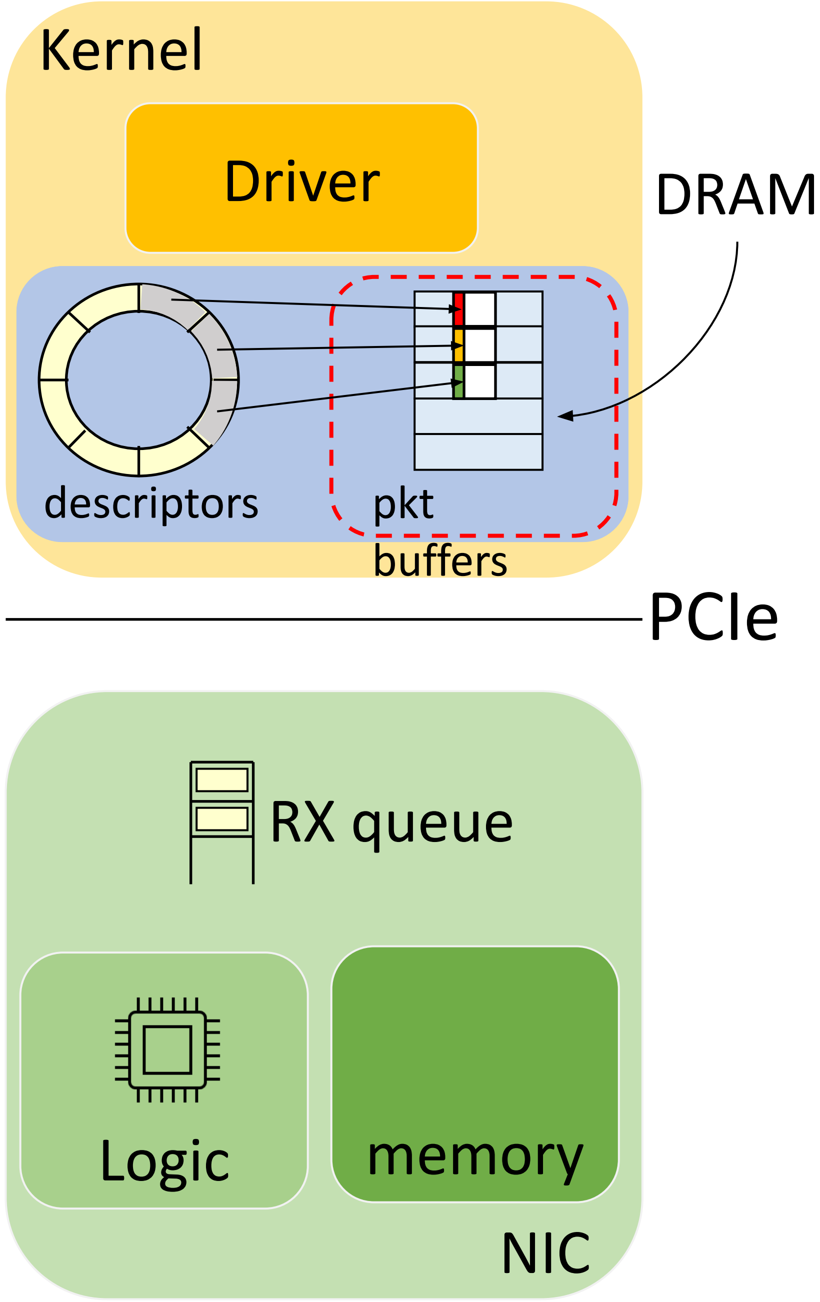


Interrupt mitigation strategies

- Modern NICs provide ways to mitigate the number of interrupts sent from HW to the CPU
  - NAPI adaptively switching between interrupt and polling according to the current workload
  - Low load: interrupt-based
  - High load: CPU-polling

Why?

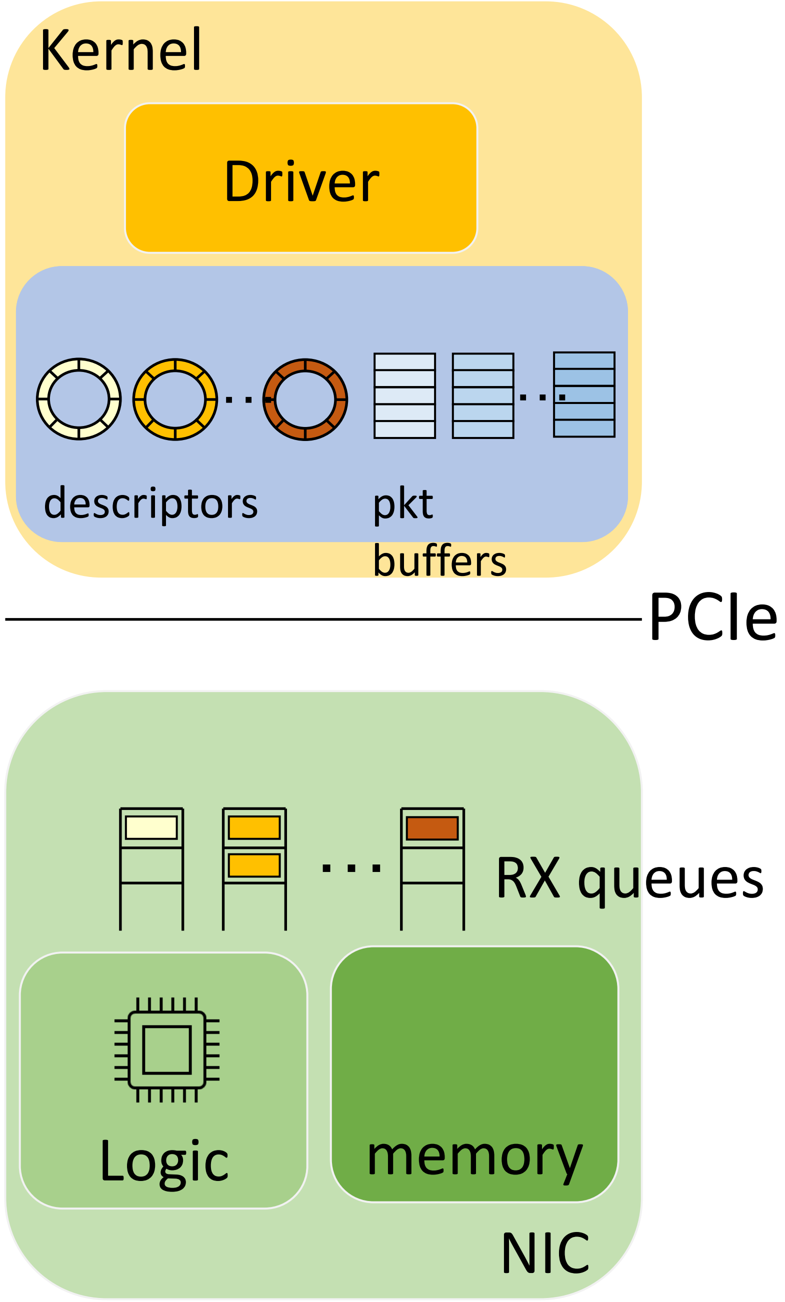
Interrupt-based guarantee low-latency processing and if the load is low, not many interrupts generated



Multi-queue

- Modern NICs provide multiple hardware queues

Why?

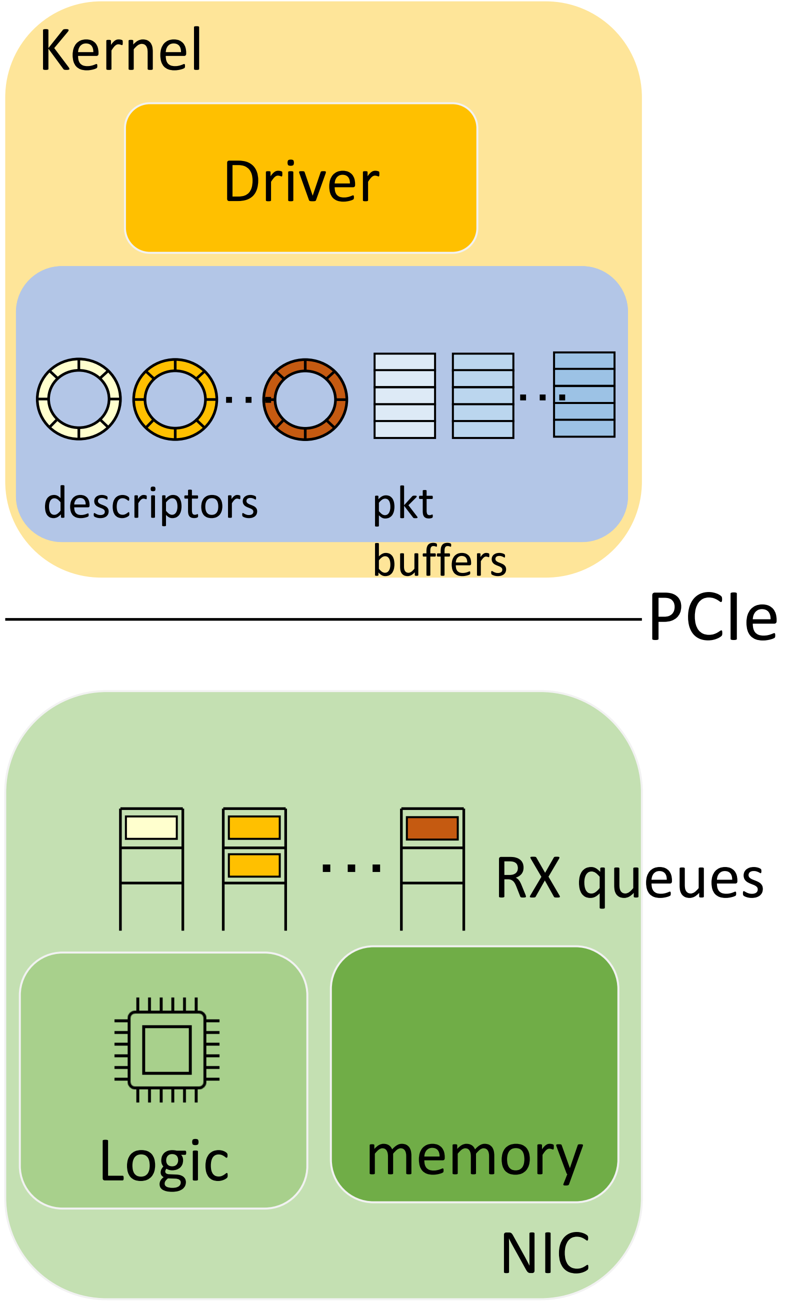


Multi-queue

- Modern NICs provide multiple hardware queues

Why?

mainly to handle concurrency between cores and performance ☺️

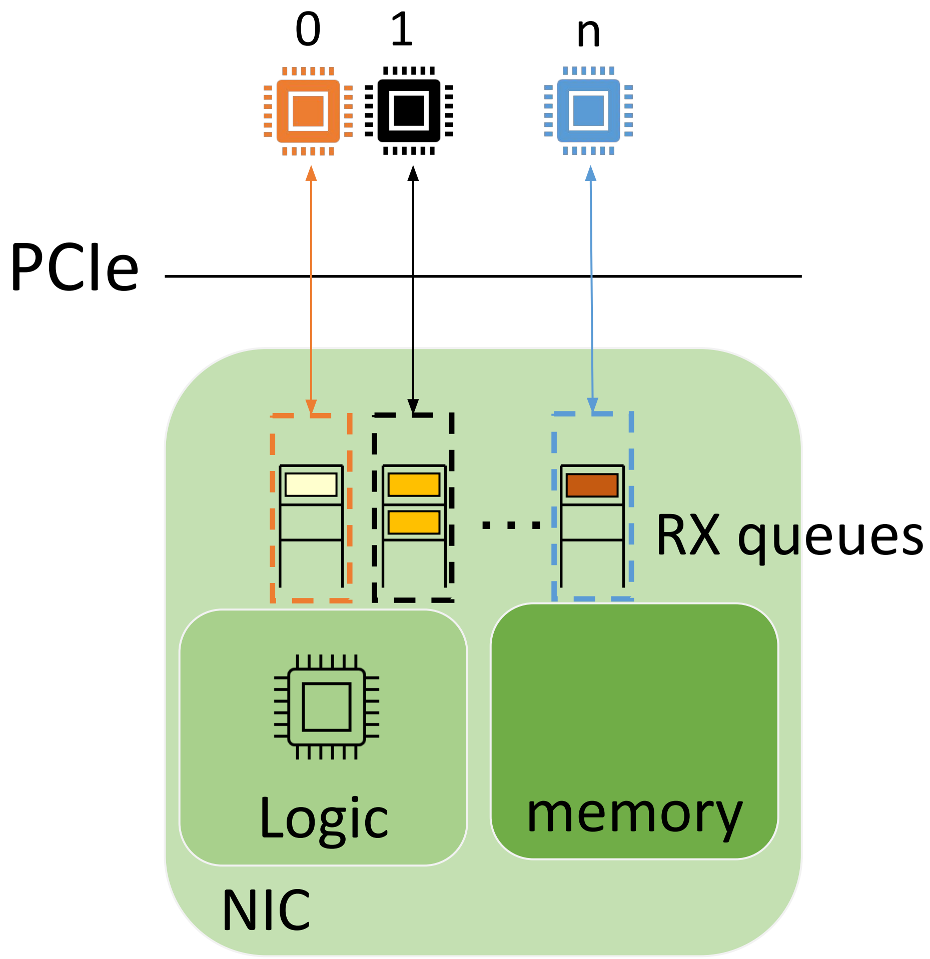


Receive-Side Scaling

- Modern NICs provide multiple hardware queues

- You can assign each queue to a CPU core and load balance the processing

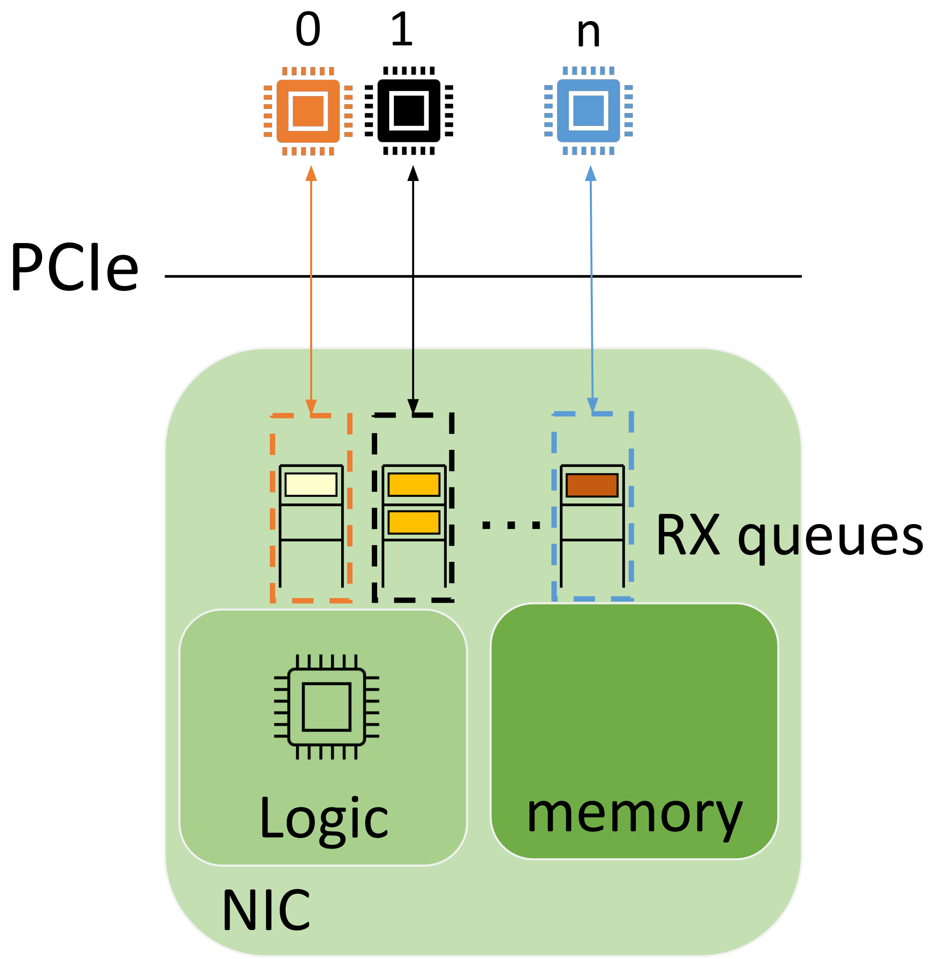
- This is called *Receive Side Scaling* which assigns flows-to-queues based on hash (IP addresses and L4 ports are common)



Receive-Side Scaling

- Modern NICs provide multiple hardware queues
- You can assign each queue to a CPU core and load balance the processing
- This is called *Receive Side Scaling* which assigns flows-to-queues based on hash (IP addresses and L4 ports are common)

Problems with this?

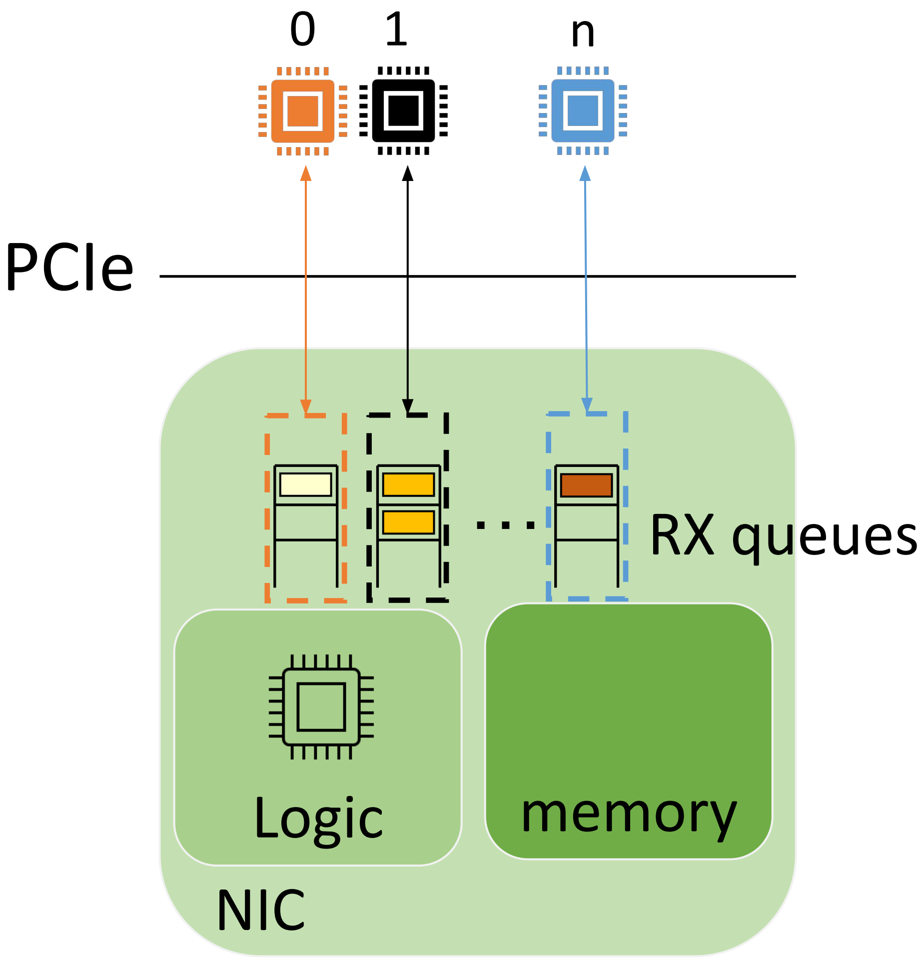


Receive-Side Scaling

- Modern NICs provide multiple hardware queues
- You can assign each queue to a CPU core and load balance the processing
- This is called *Receive Side Scaling* which assigns flows-to-queues based on hash (IP addresses and L4 ports are common)

Problems with this?

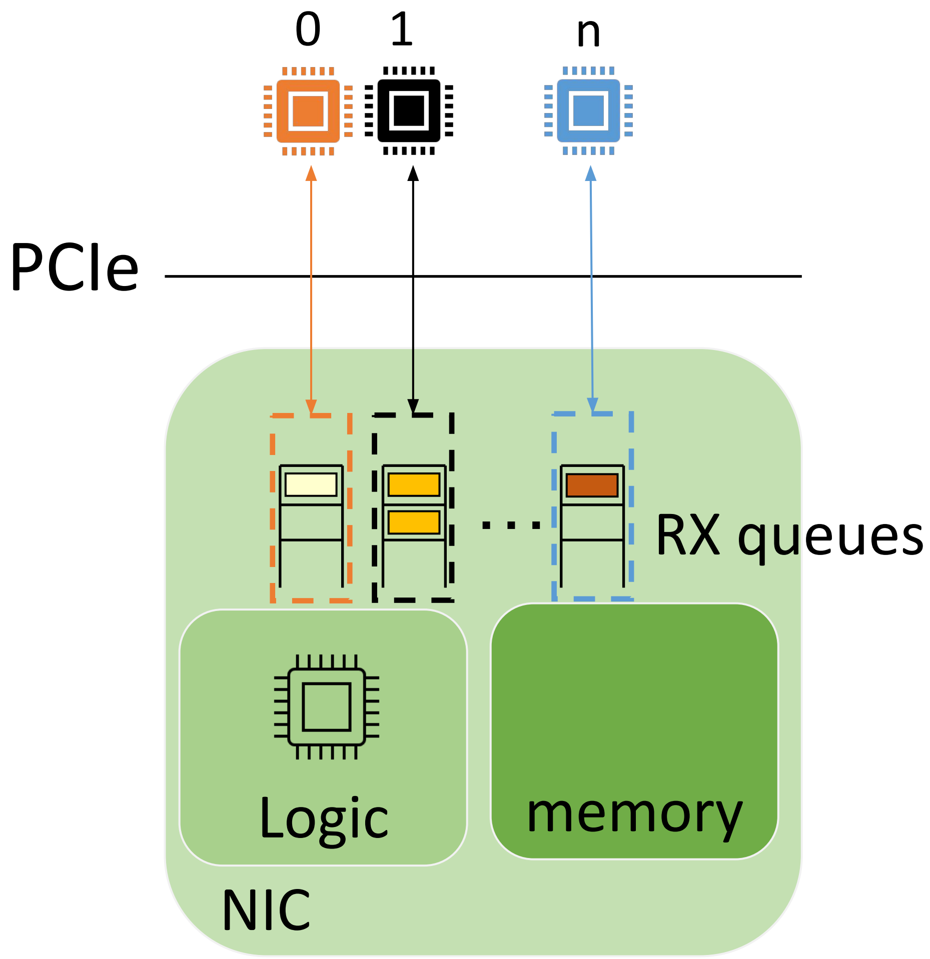
Hash imbalance



Receive-Side Scaling

- RSS tries to solve the problem of the overloaded core receiving all network traffic

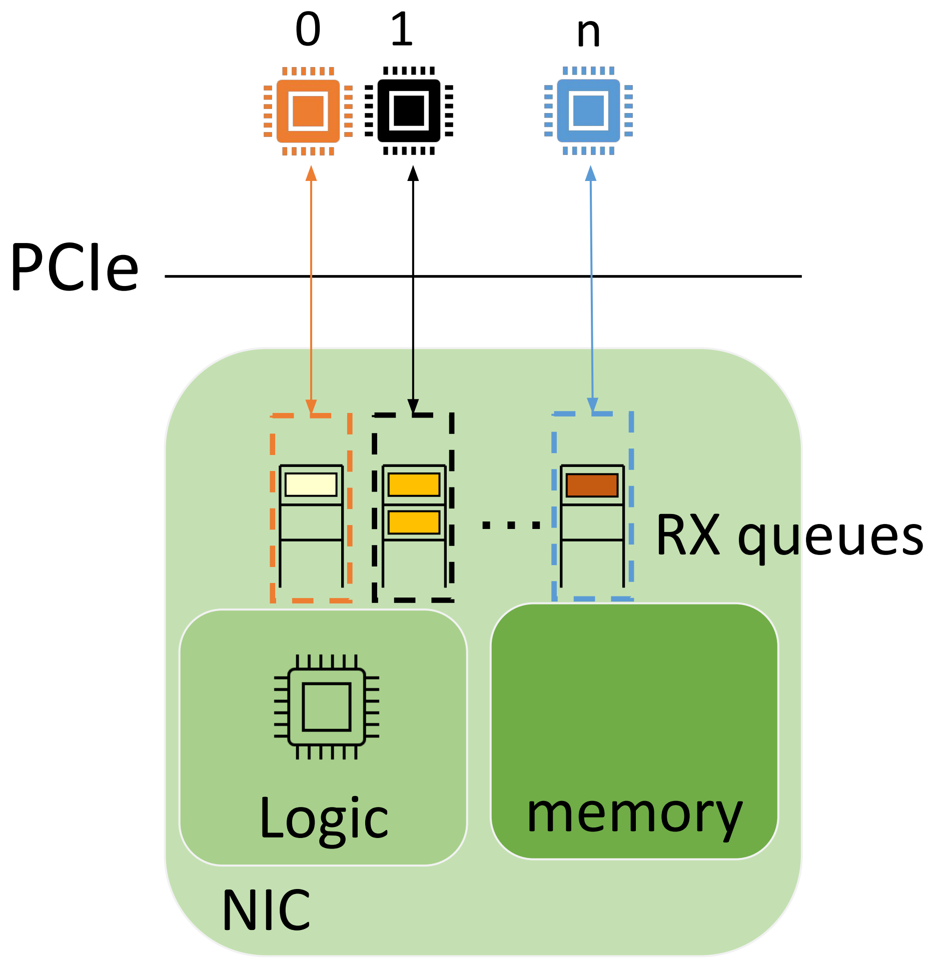
- When using RSS packets are sent to different cores for interrupt processing



Receive-Side Scaling

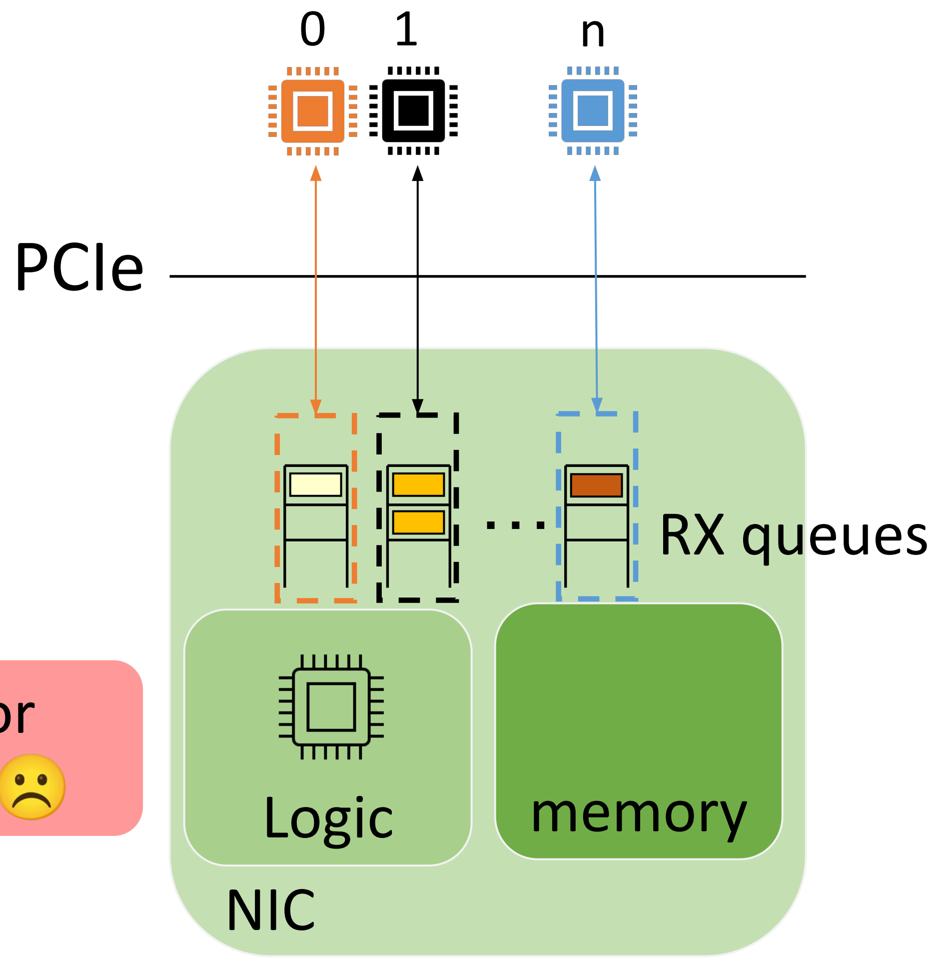
- RSS tries to solve the problem of the overloaded core receiving all network traffic
- When using RSS packets are sent to different cores for interrupt processing

Note: application consuming the packet might be on a different core with respect to the one selected by RSS



Receive-Side Scaling

- RSS tries to solve the problem of the overloaded core receiving all network traffic
- When using RSS packets are sent to different cores for interrupt processing

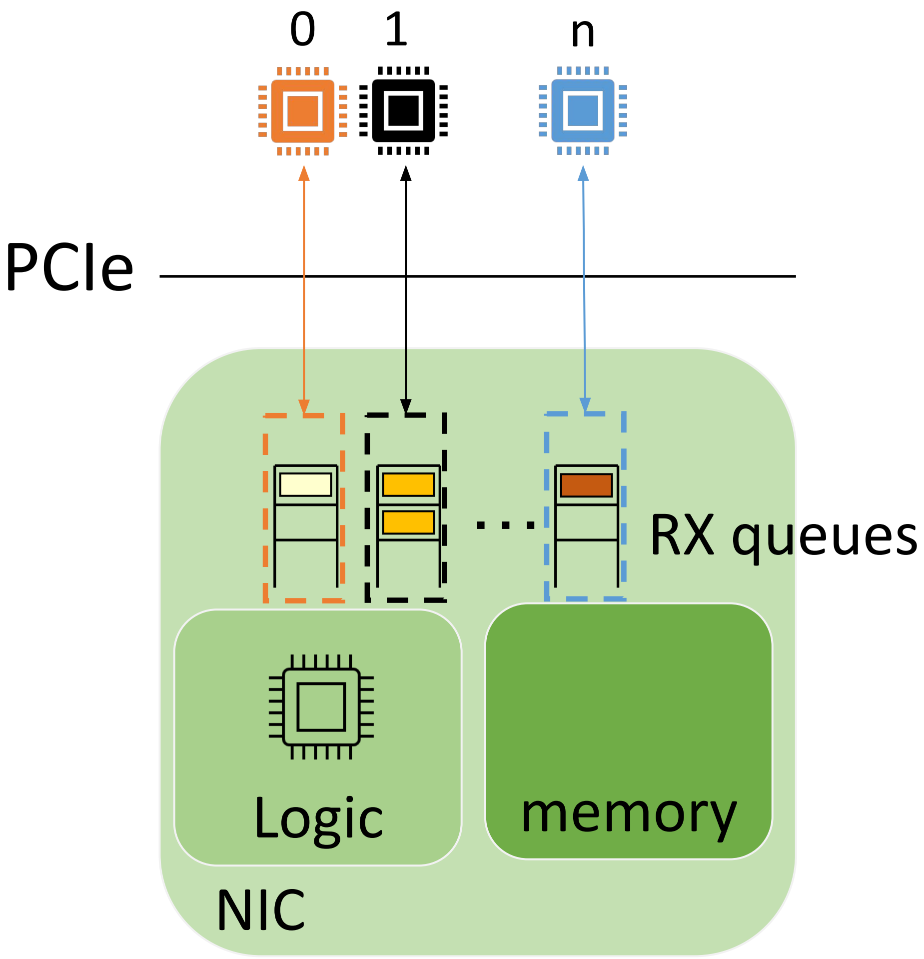


Note: application consuming the packet might be on a different core with respect to the one selected by RSS

This is bad for performance ☹

Accelerated Receive-Flow Steering

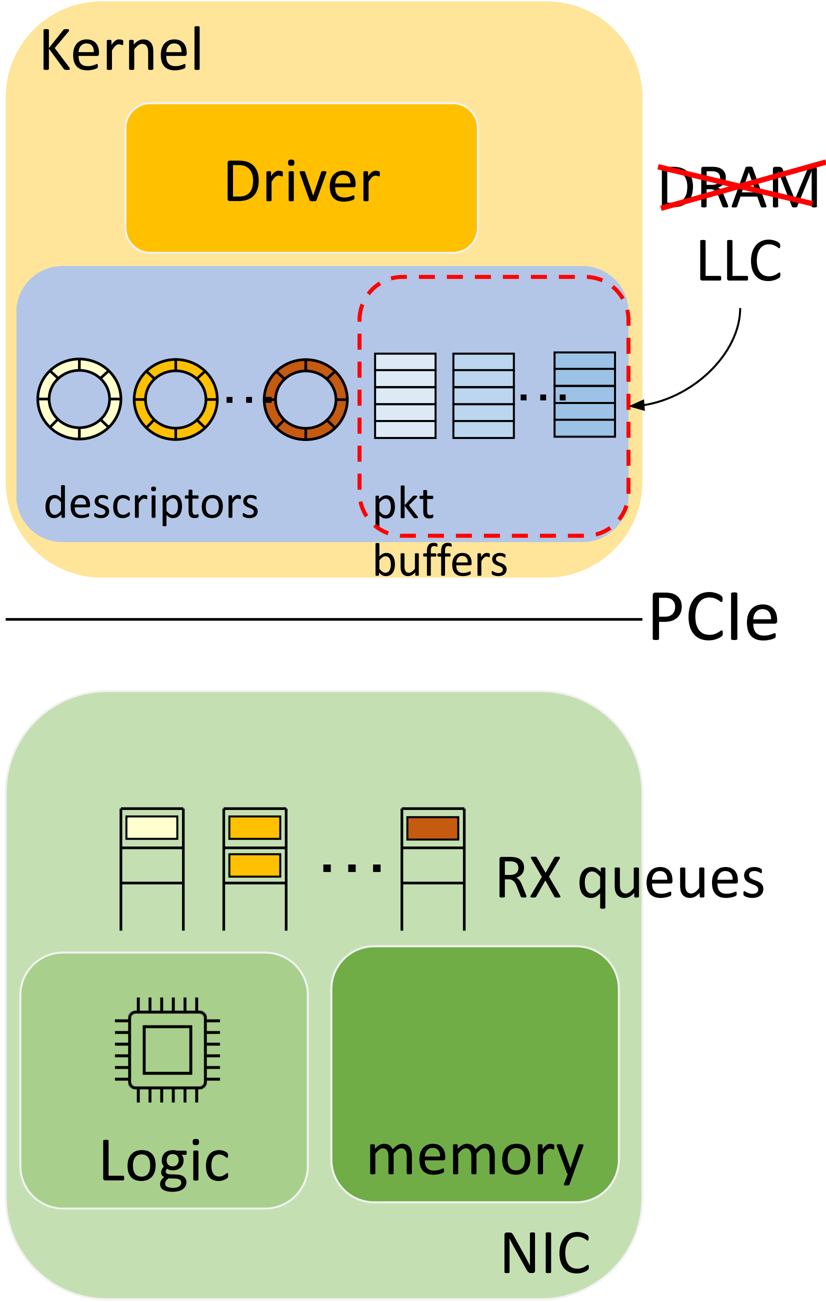
- aRFS allows you to direct packets to the same core where an application is running
- Example: Intel Flow-Director supports advanced filters that *direct* received packets to different queues and enables tight control on flow in the platform.
- Flow-Director matches flows and the CPU core where the application is running, and provide multiple parameters for flow classification and load balancing



Data Direct I/O

- Modern CPUs provide the possibility to directly store the packet in L3 cache (LLC) (e.g., Intel DDIO)

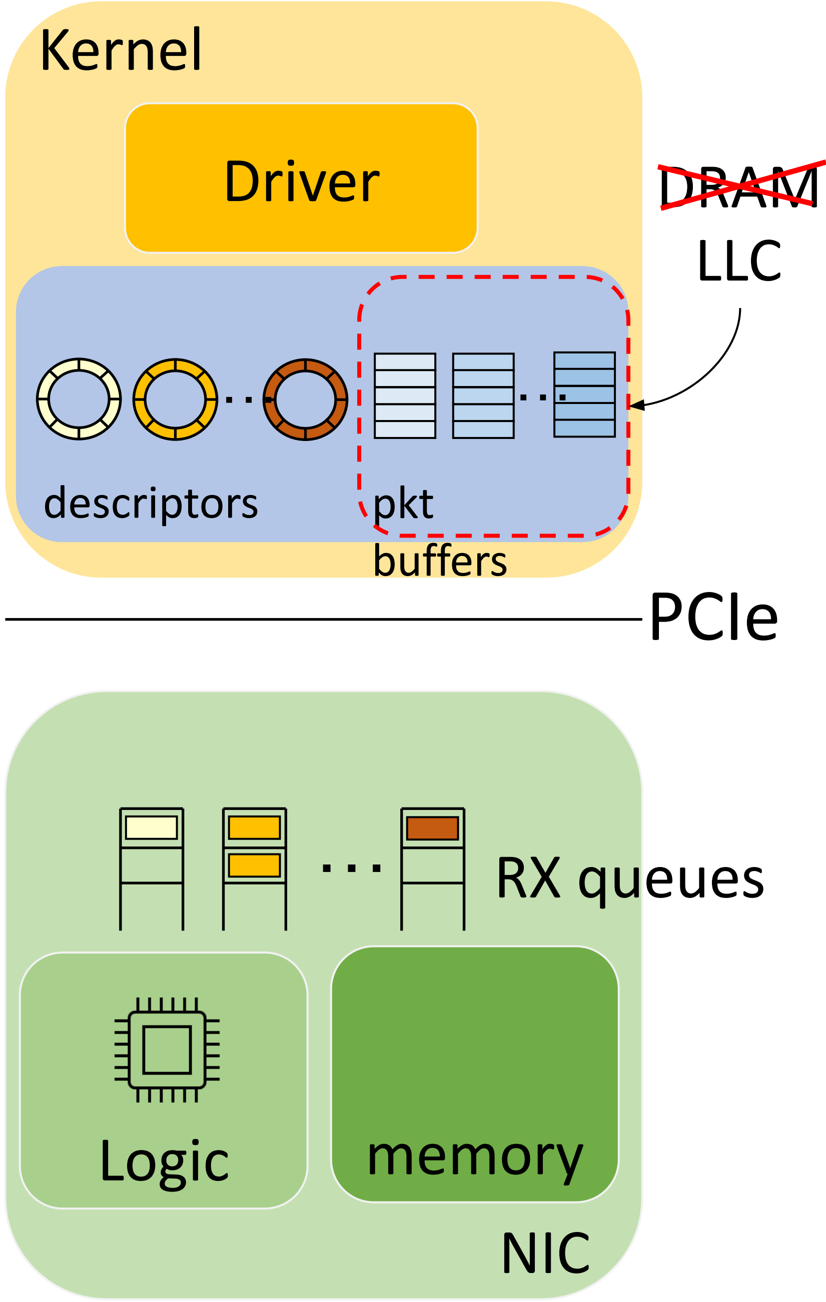
- The CPU core does not have anymore to move data from DRAM to cache when processing the packet



Data Direct I/O

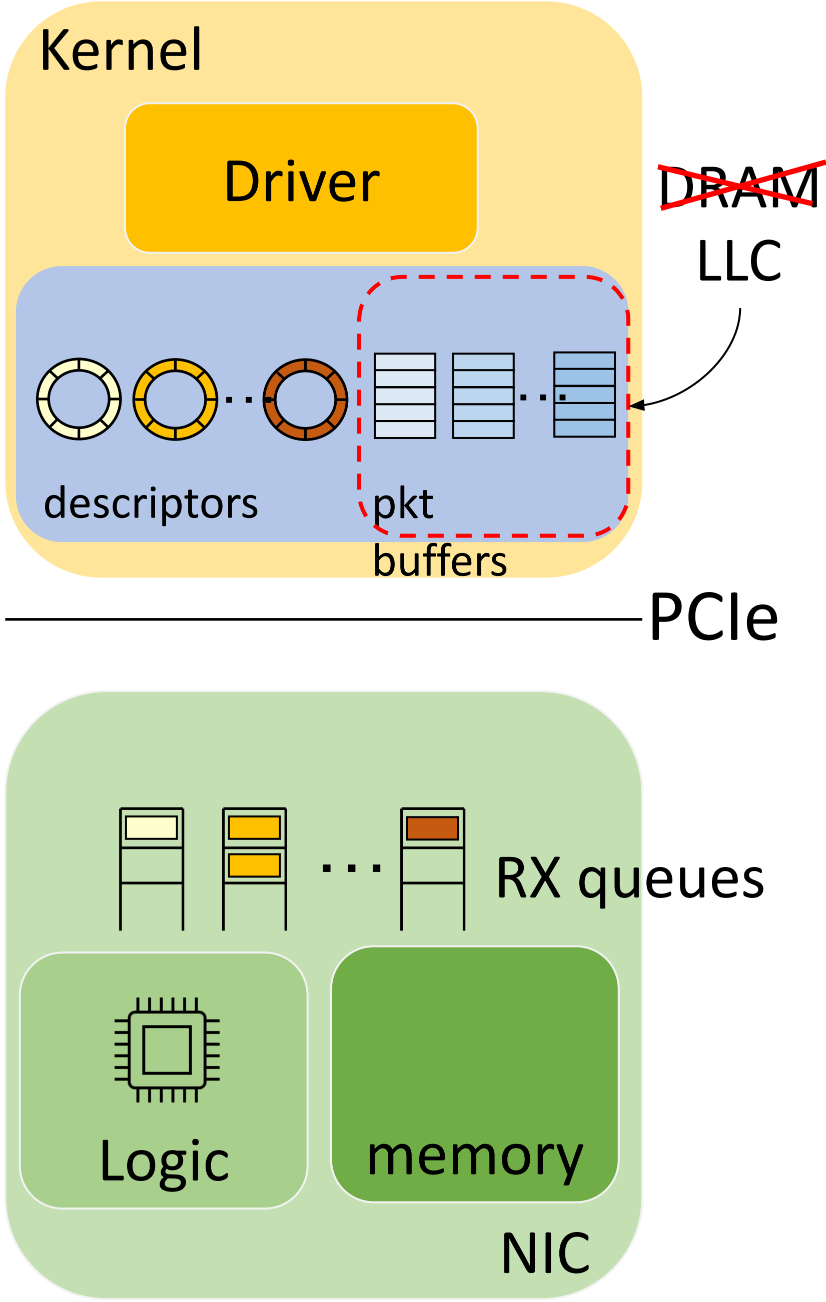
- Modern CPUs provide the possibility to directly store the packet in L3 cache (LLC) (e.g., Intel DDIO)
- The CPU core does not have anymore to move data from DRAM to cache when processing the packet

Problems with this?



Data Direct I/O

- Modern CPUs provide the possibility to directly store the packet in L3 cache (LLC) (e.g., Intel DDIO)
- The CPU core does not have anymore to move data from DRAM to cache when processing the packet

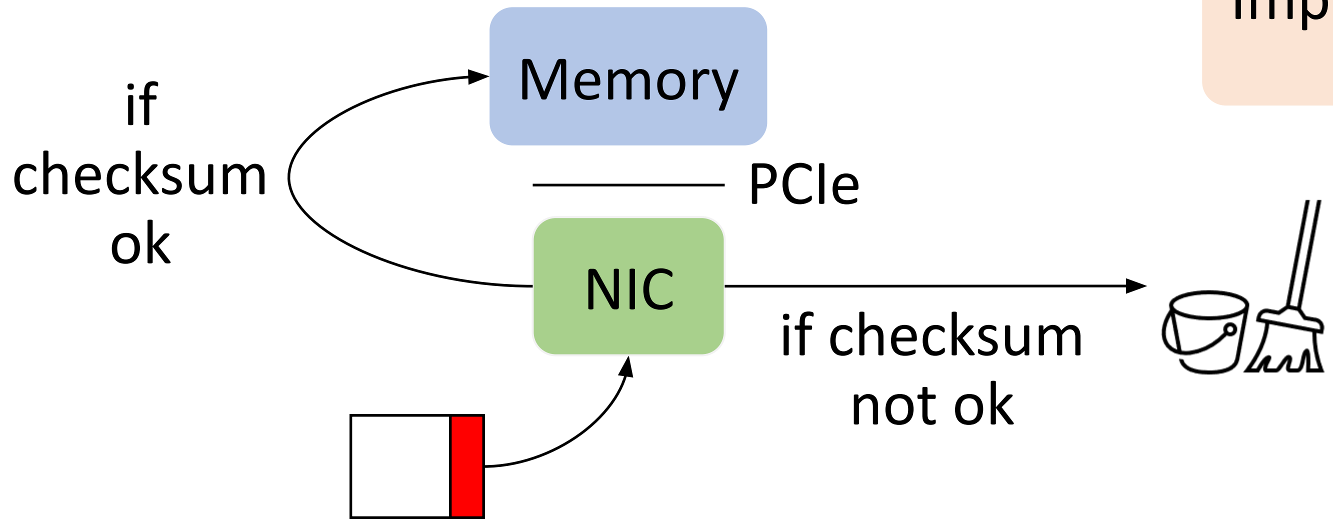


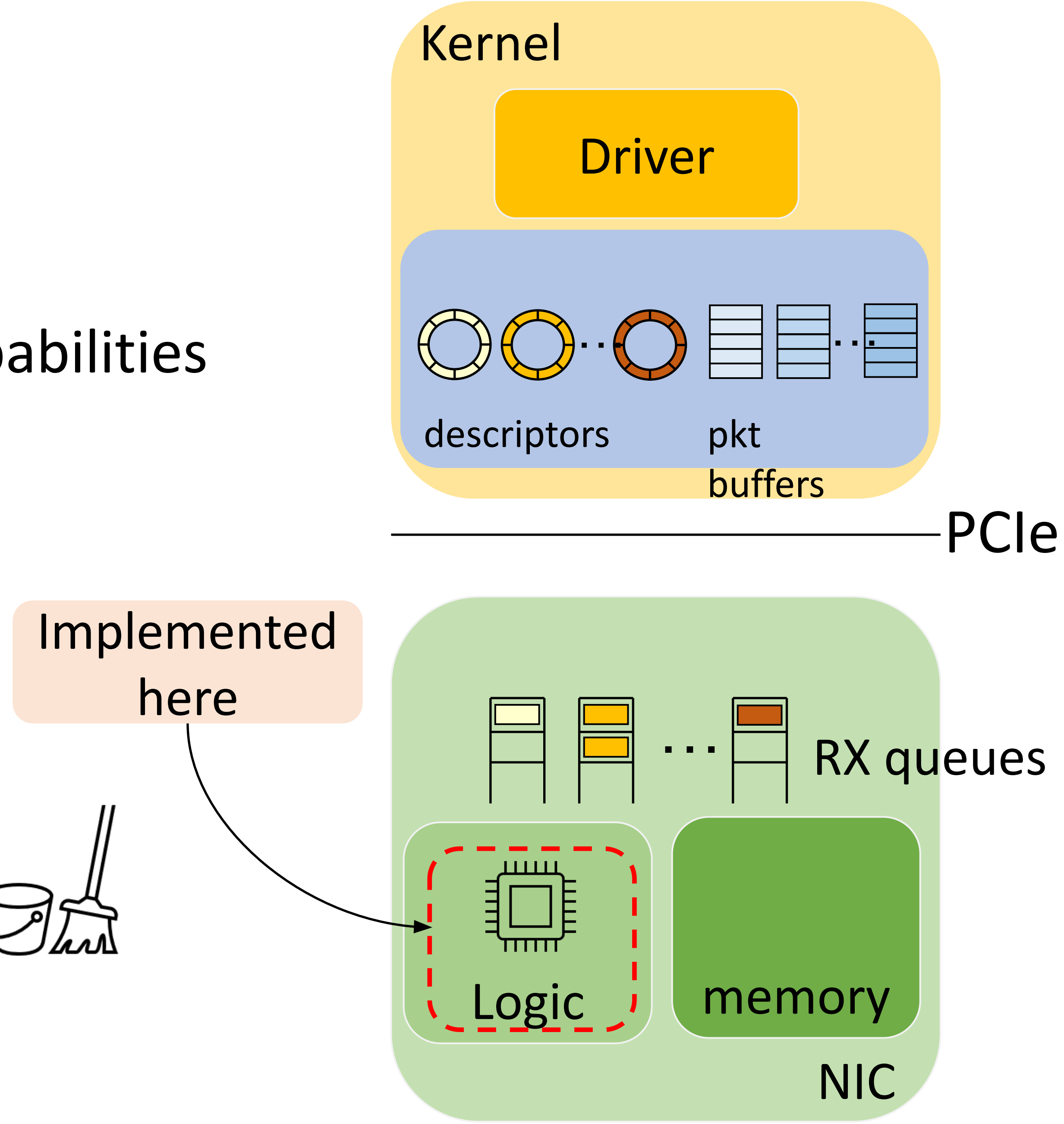
Problems with this?

new incoming packets repeatedly evict not-yet-processed packets from the LLC: this is called the *leaky DMA problem*

Standard offloads

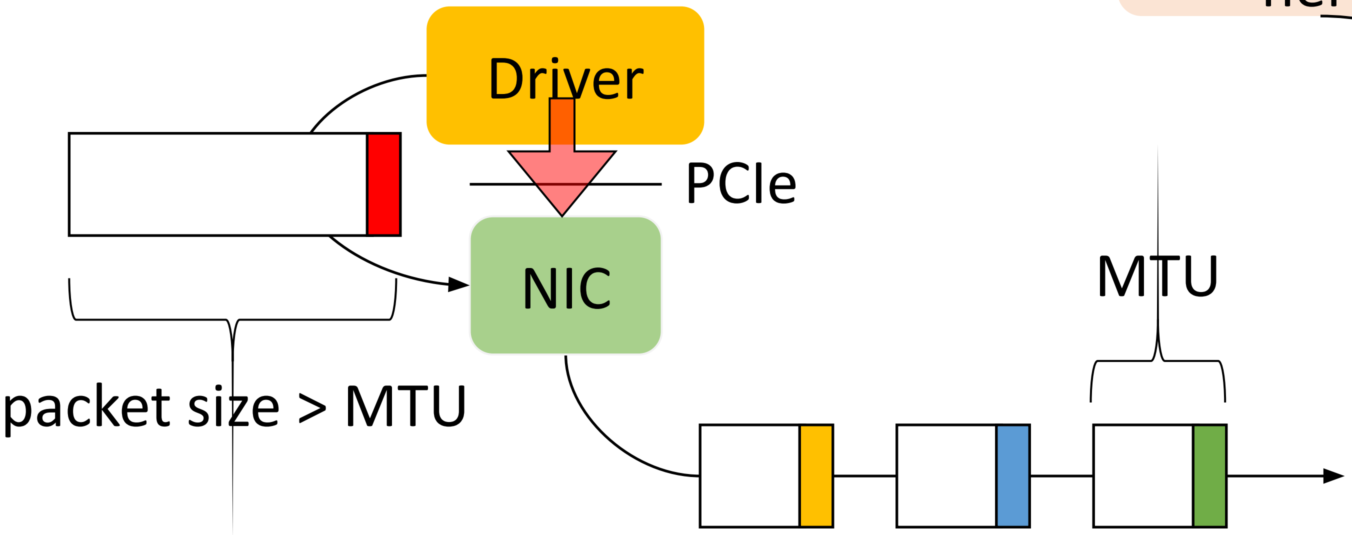
- Modern NICs provide some offload capabilities in the form of:
  - Checksum calculation

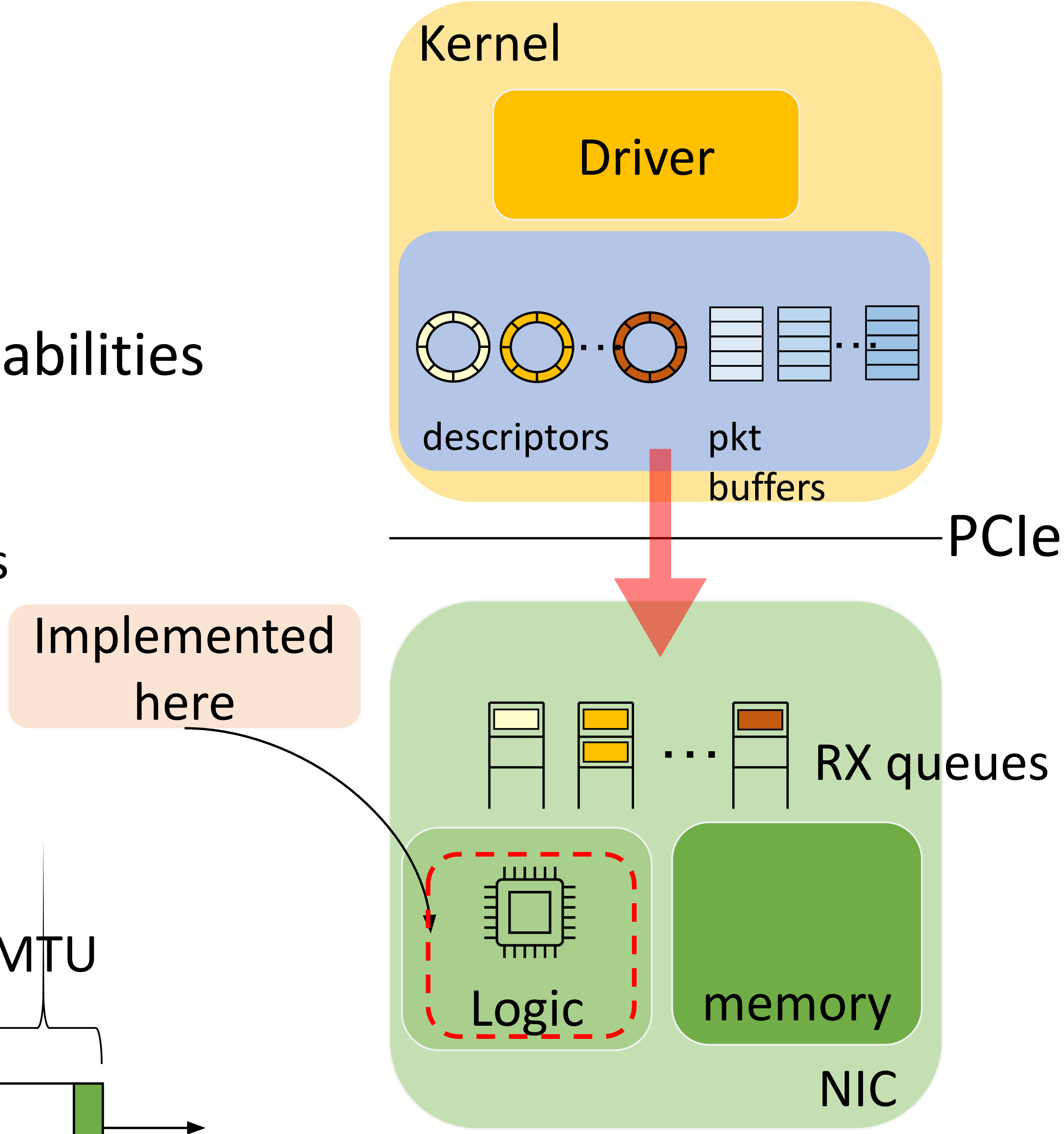




Standard offloads

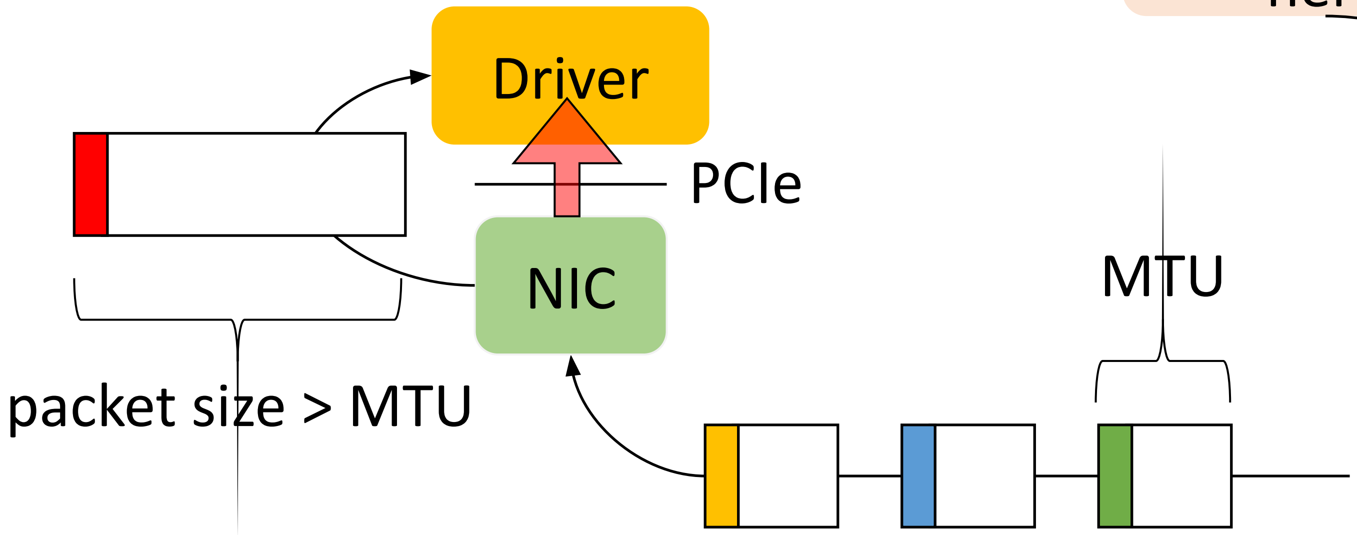
- Modern NICs provide some offload capabilities in the form of:
  - TCP Segmentation Offload (TSO) and UDP Fragmentation Offload (UFO): NIC handles segmentation/fragmentation

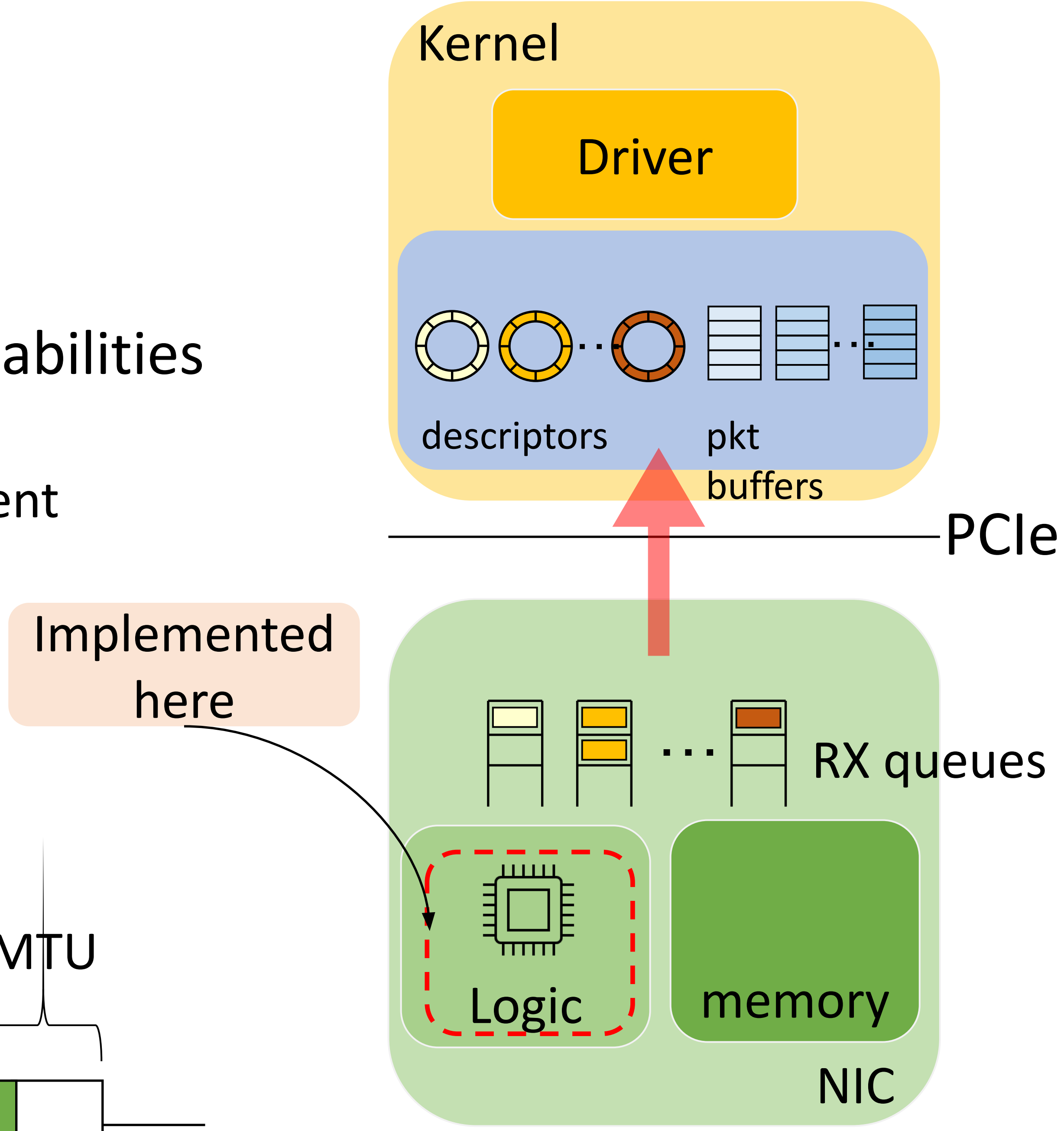




Standard offloads

- Modern NICs provide some offload capabilities in the form of:
  - Large Receive Offload (LRO): NIC re-segment incoming packets

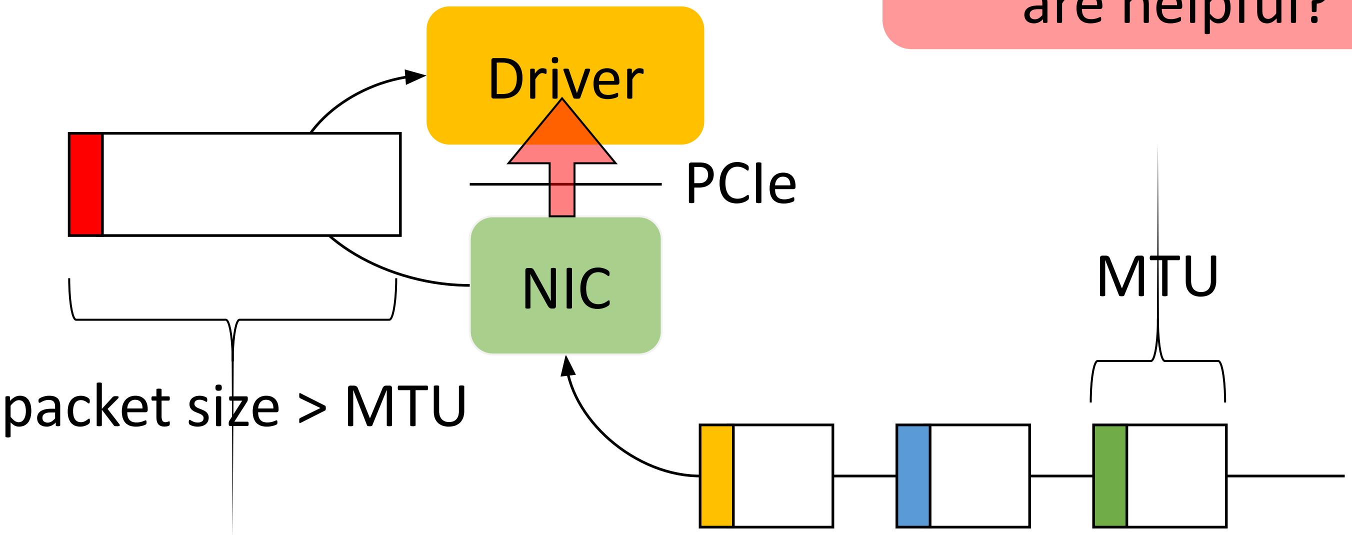


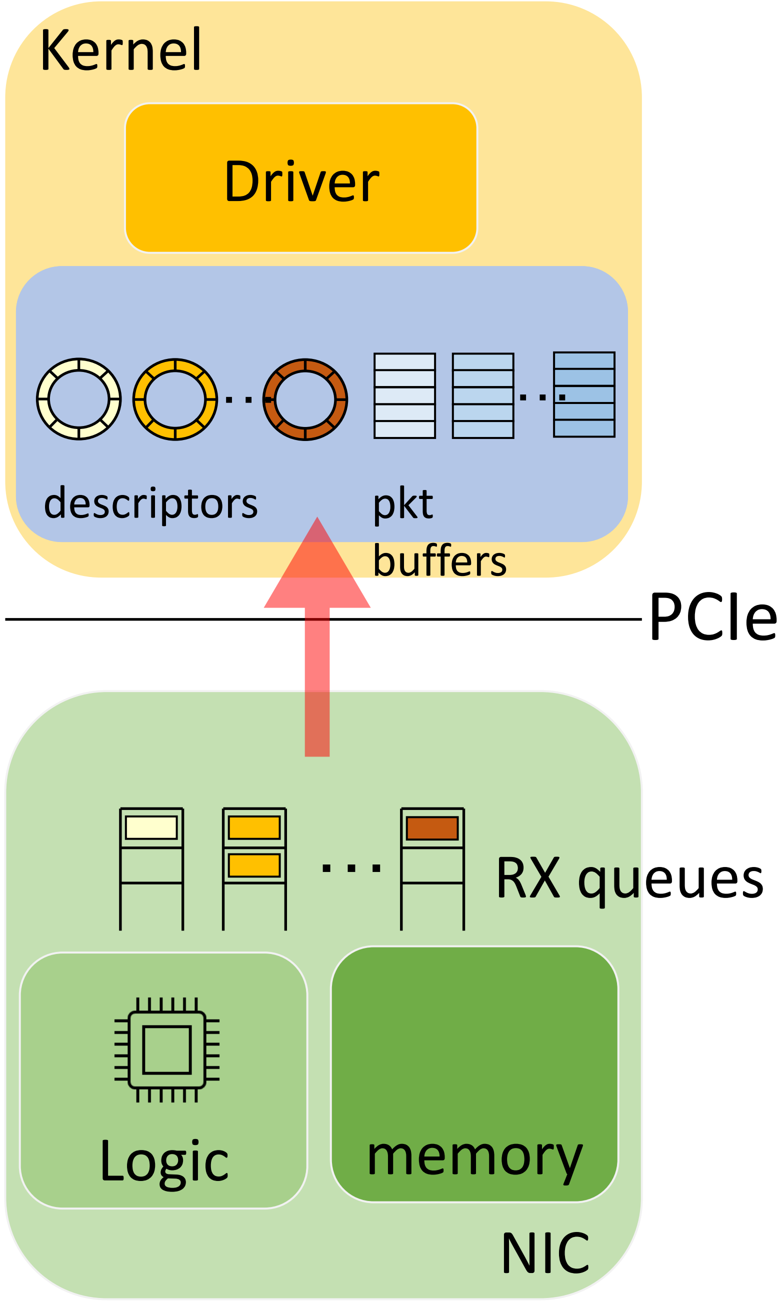


Standard offloads

- Modern NICs provide some offload capabilities in the form of:
  - Large Receive Offload (LRO): NIC re-segment incoming packets

Why TSO, UFO, LRO are helpful?



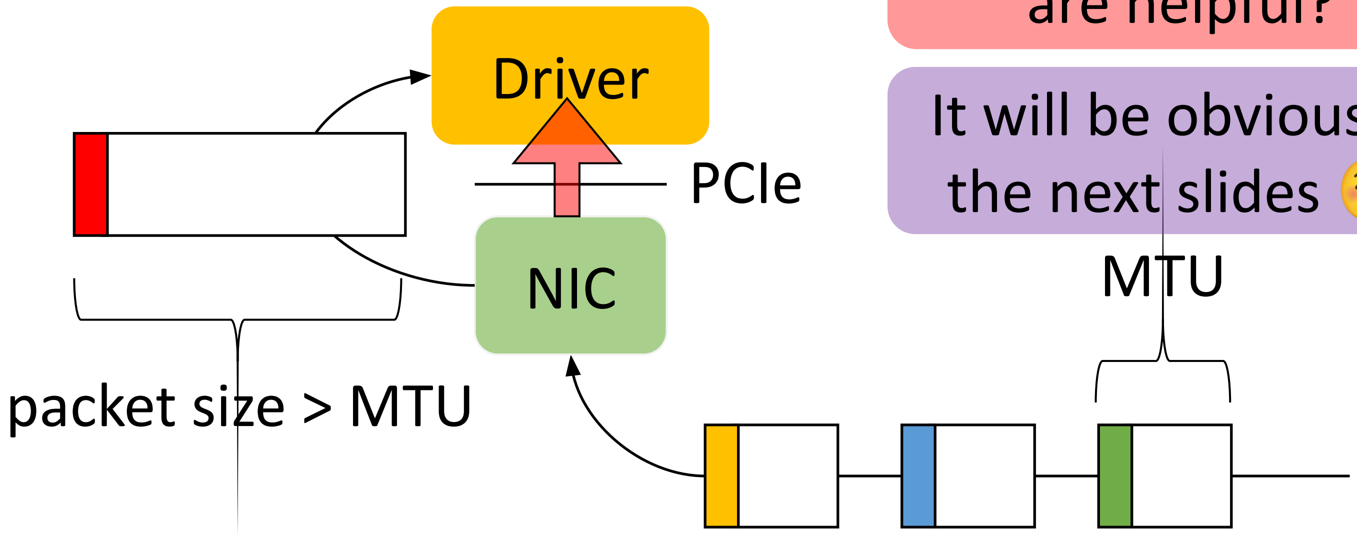


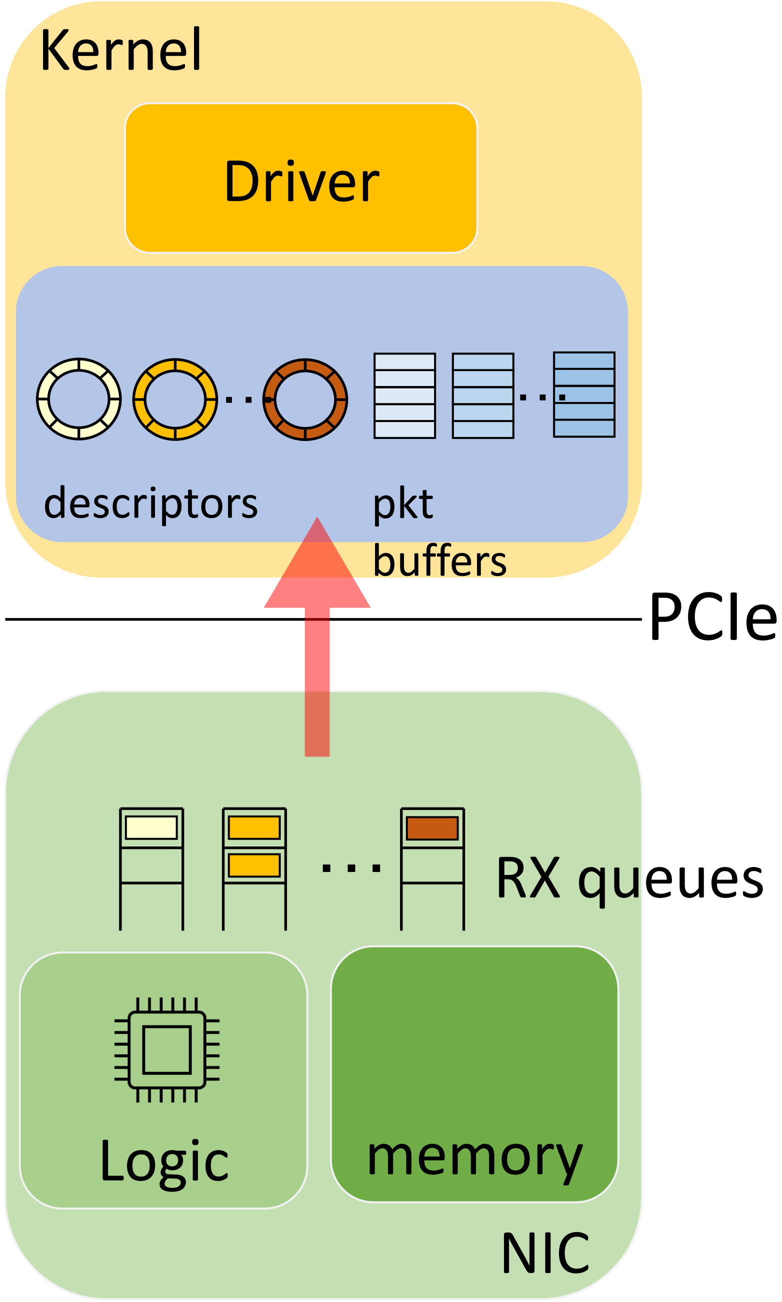
Standard offloads

- Modern NICs provide some offload capabilities in the form of:
  - Large Receive Offload (LRO): NIC re-segment incoming packets

Why TSO, UFO, LRO are helpful?

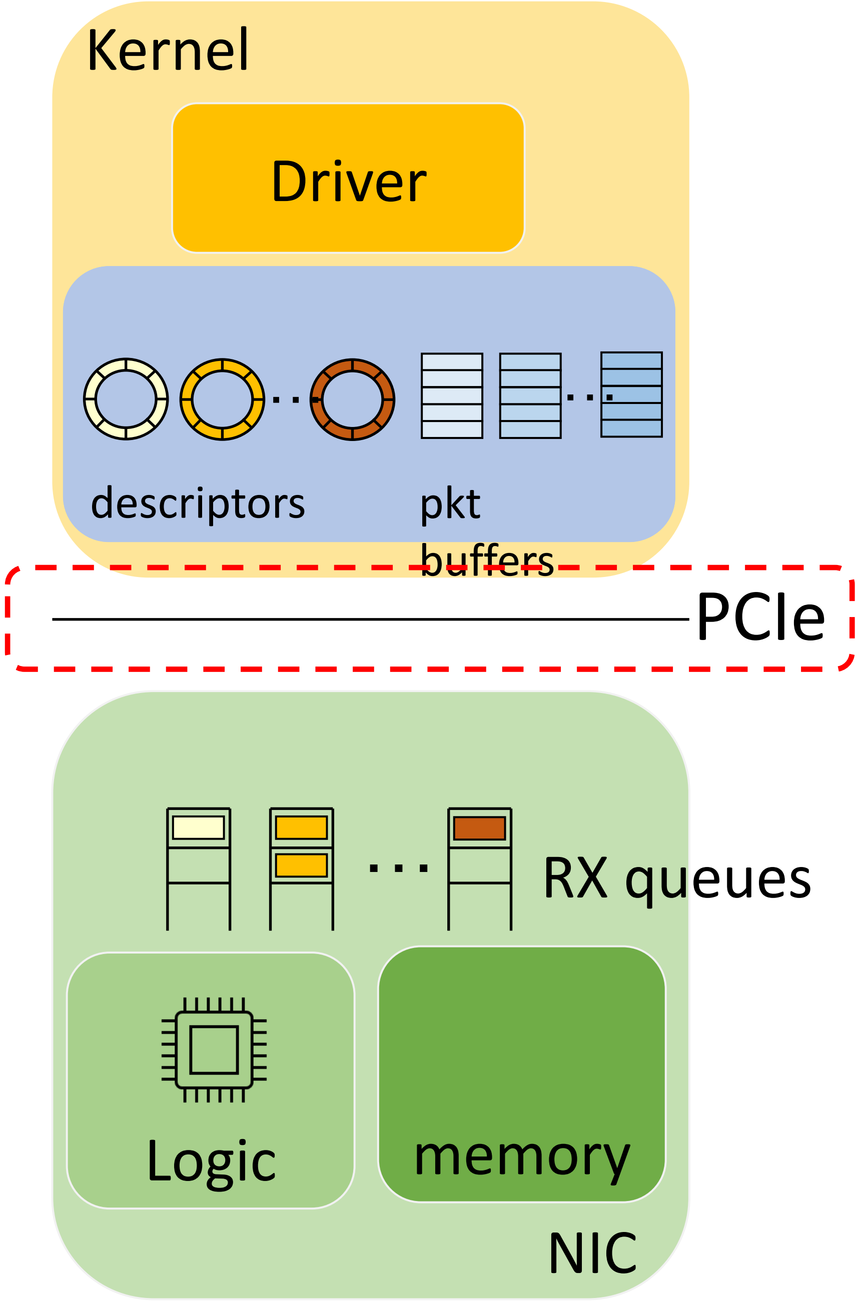
It will be obvious in the next slides ☺️





PCIe

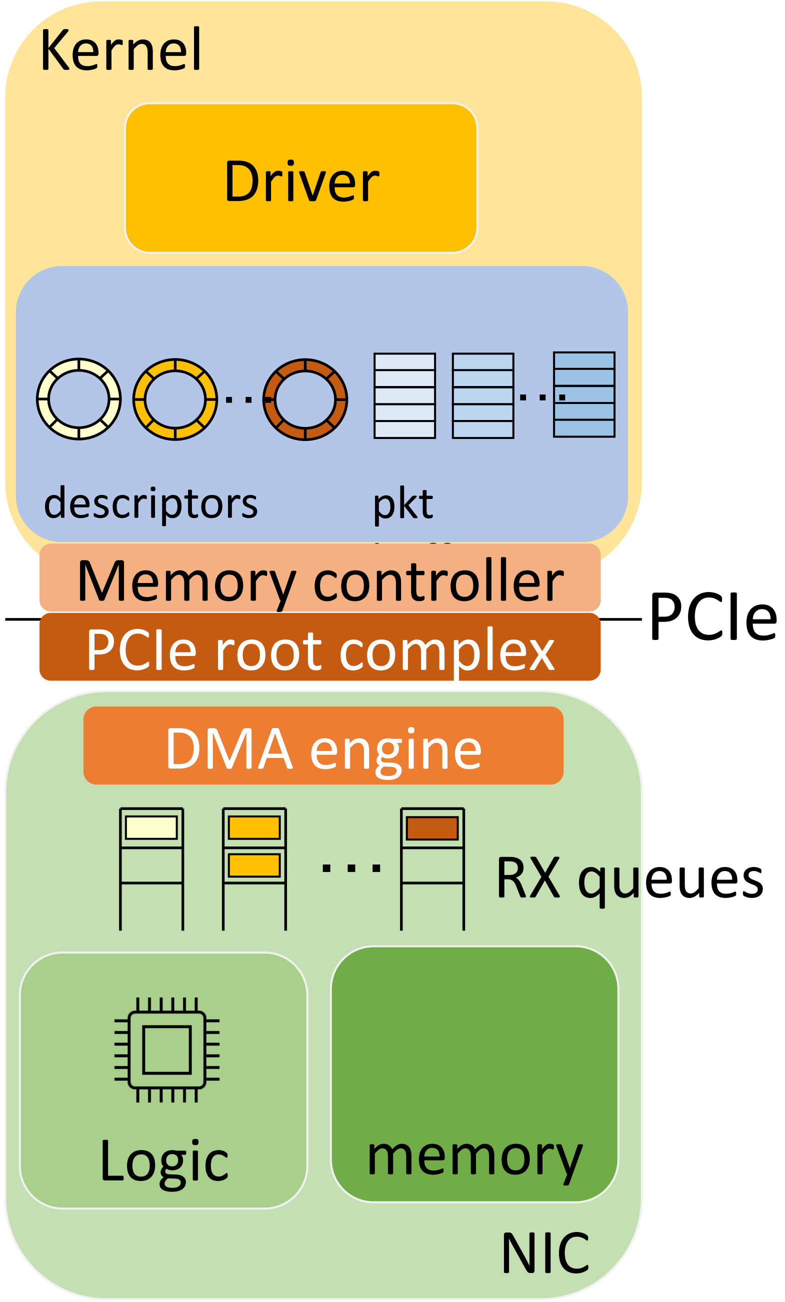
- All the operations discussed so far require PCIe transactions
  - Packet data from NIC to host memory
  - descriptors
  - IRQs
- PCIe is the de-facto standard to connect high performance devices to the rest of the system



PCIe

- All the operations discussed so far require PCIe transactions
    - Packet data from NIC to host memory
    - descriptors
    - IRQs

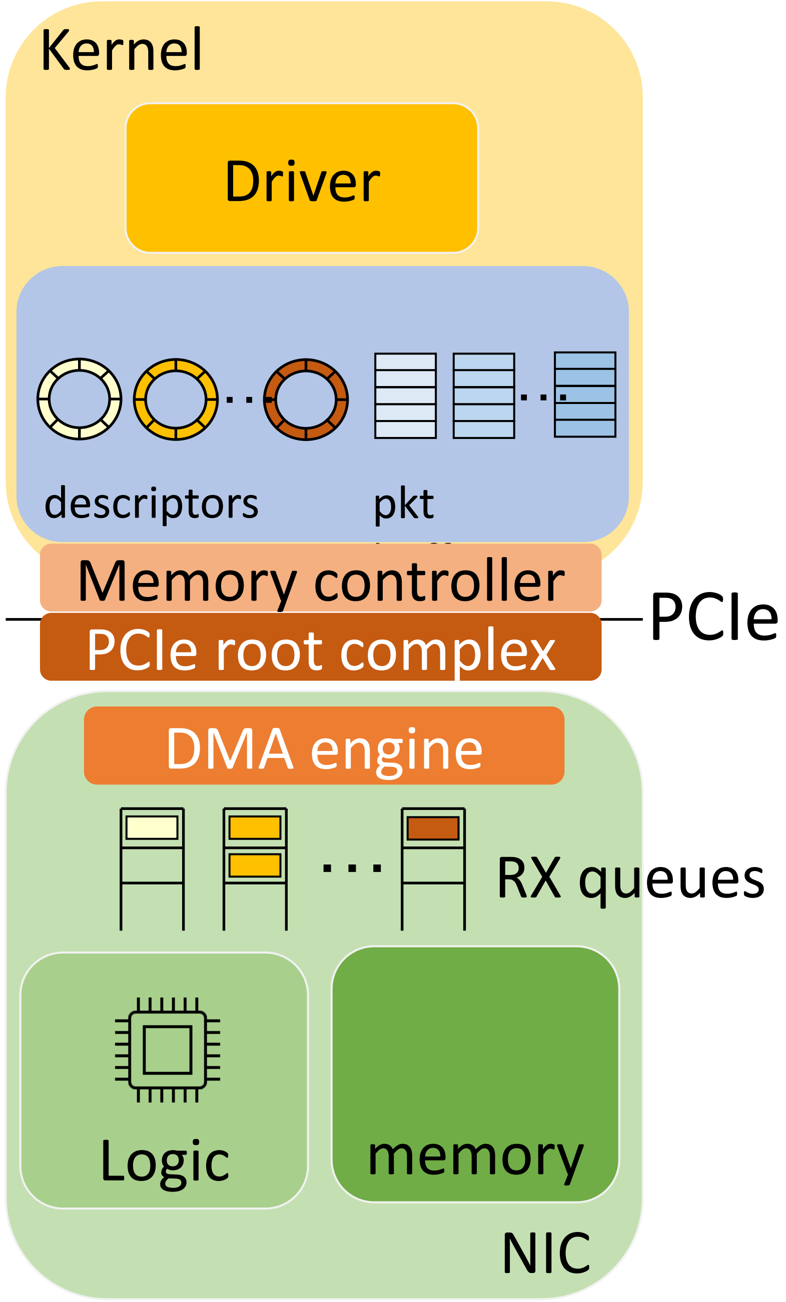
- PCIe is the de-facto standard to connect high performance devices to the rest of the system



PCIe

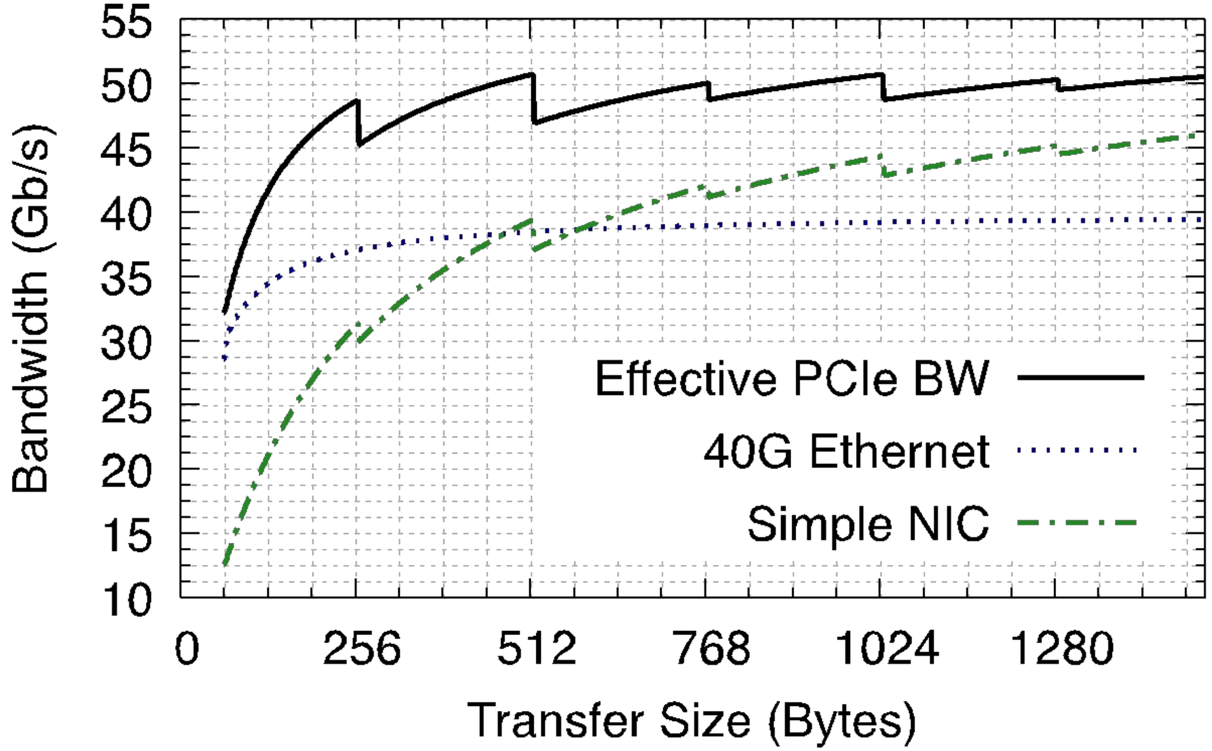
- When you DMA something (e.g., write X to host at address Y), the DMA engine breaks the request in multiple PCIe Memory Write packets (*Transaction Layer Packets*)

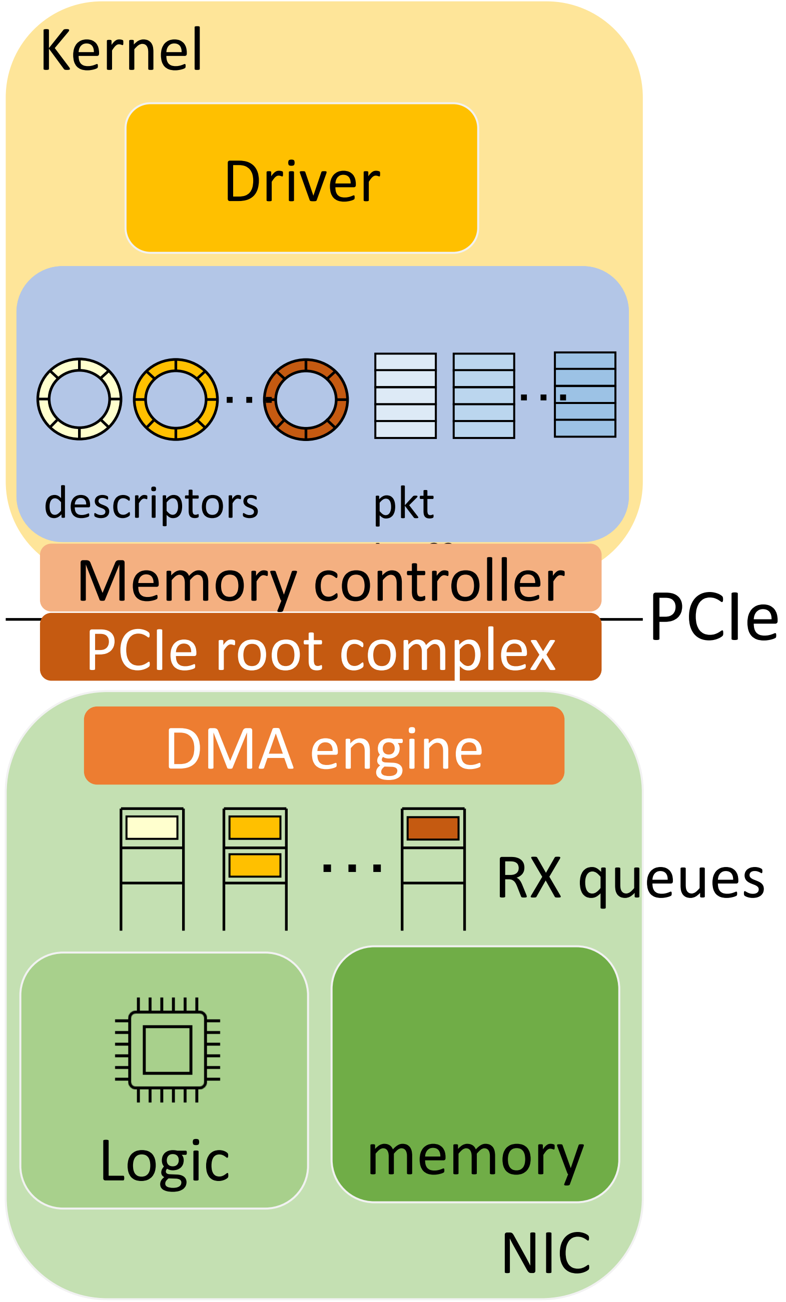
- PCIe is almost like a network protocol with packets (TLPs), headers, MTU, flow control, addressing etc..



The impact of PCIe

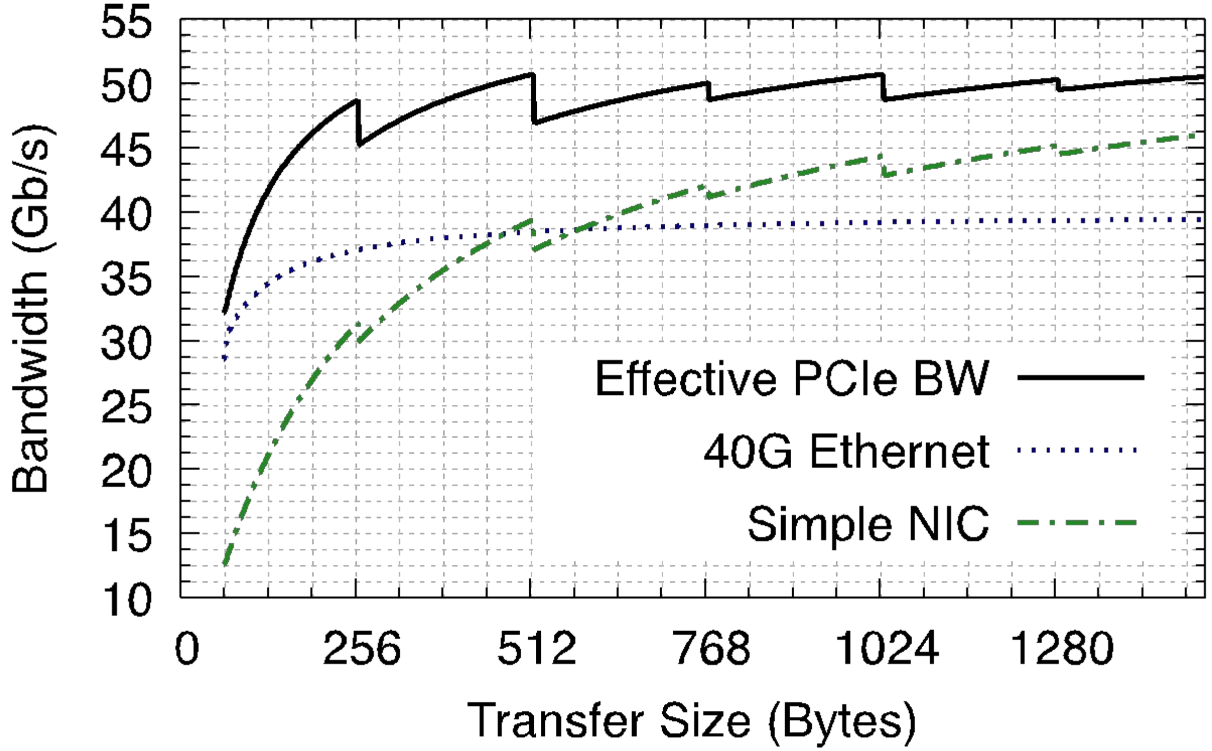
PCIe Gen 3 x8 (typically used by 40Gbps NICs)

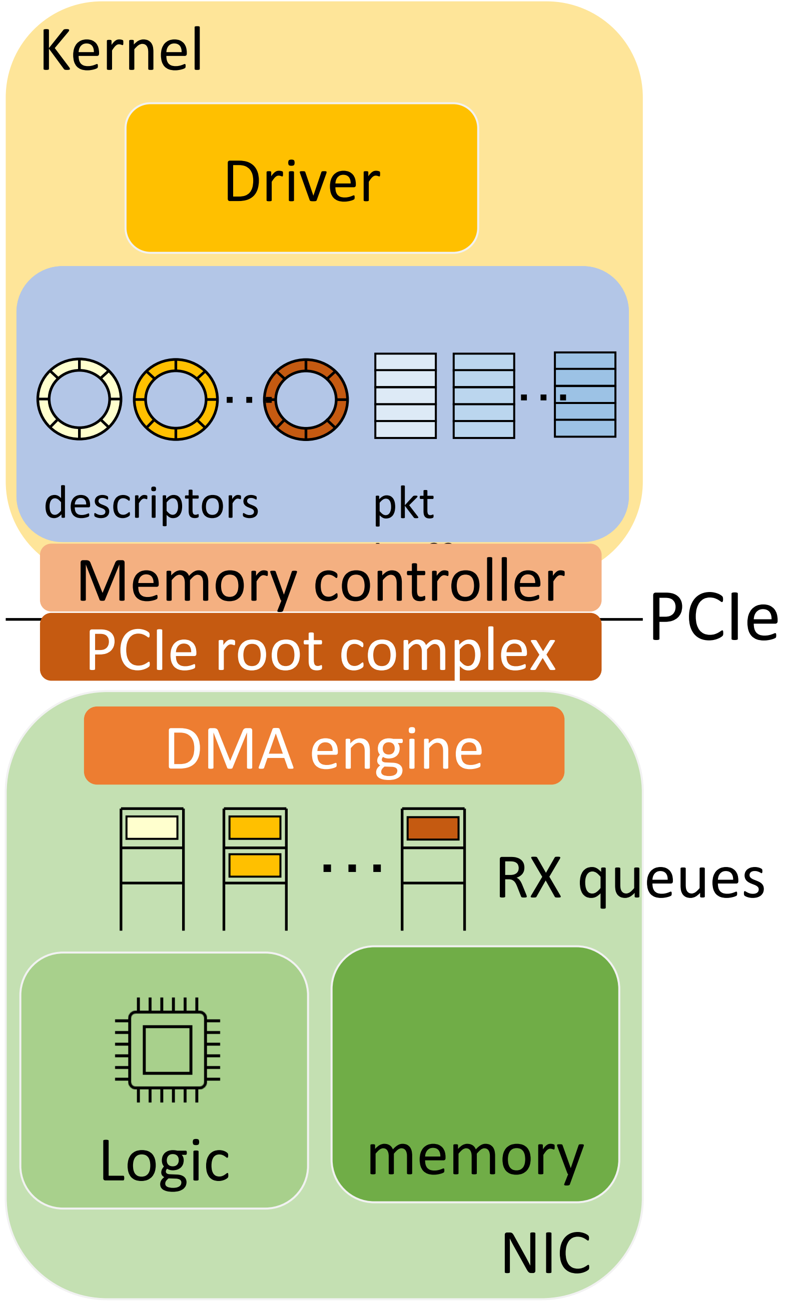




At the physical layer it would have 62.96Gbps of throughput but the PCIe protocol reduces the usable bandwidth to approx 50Gbps

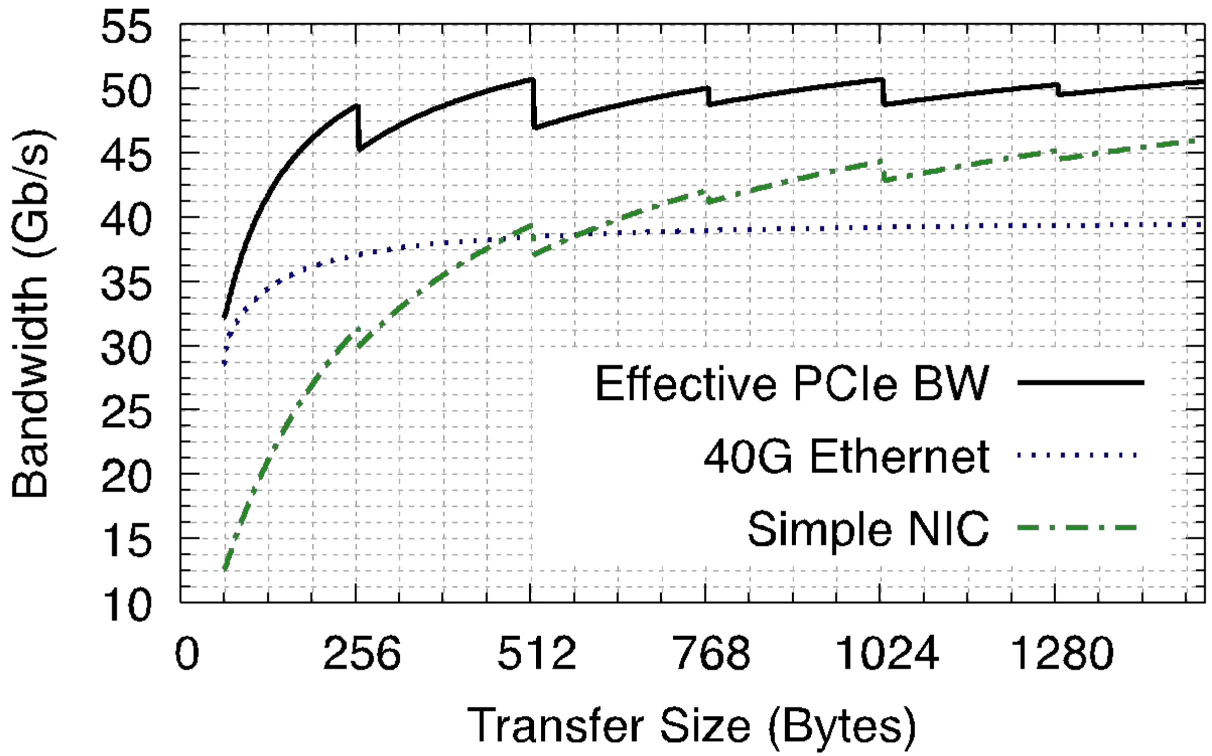
The impact of PCIe

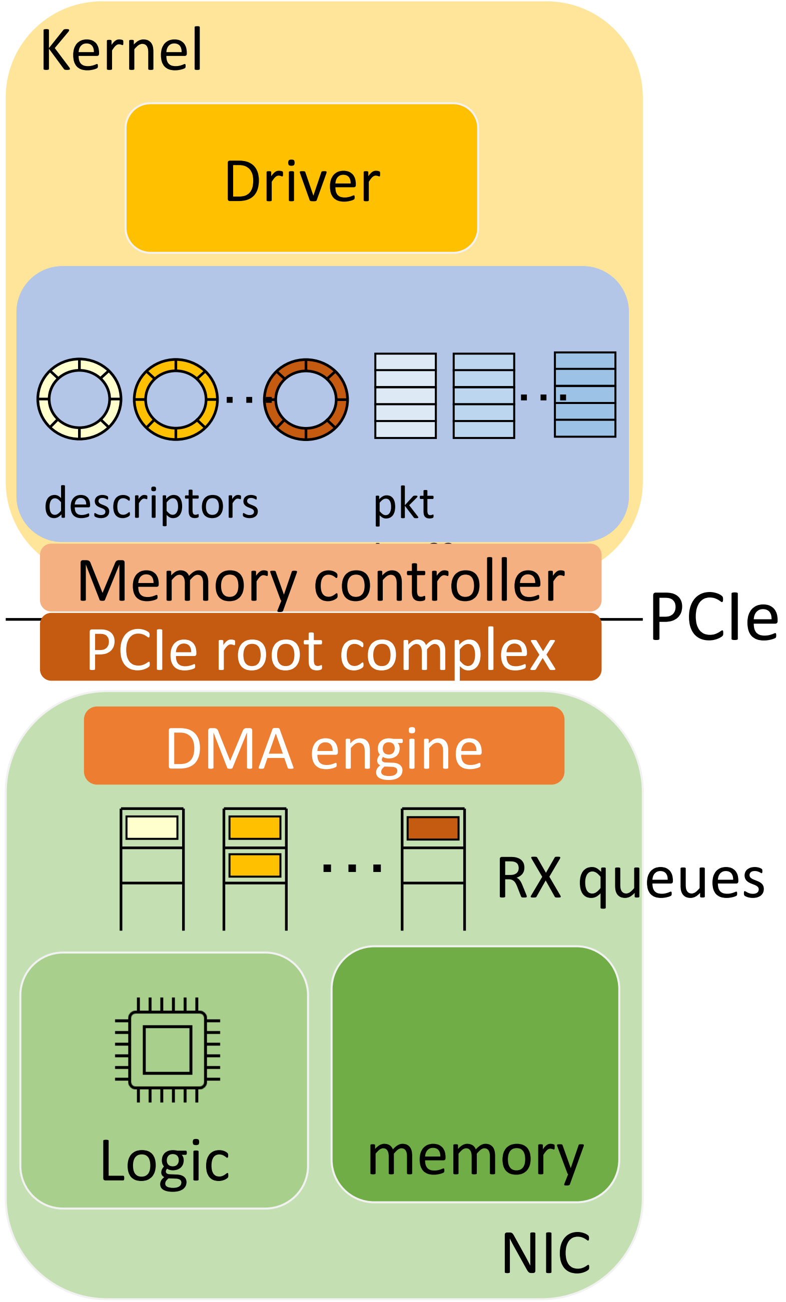




Why the sawtooth pattern?

The impact of PCIe



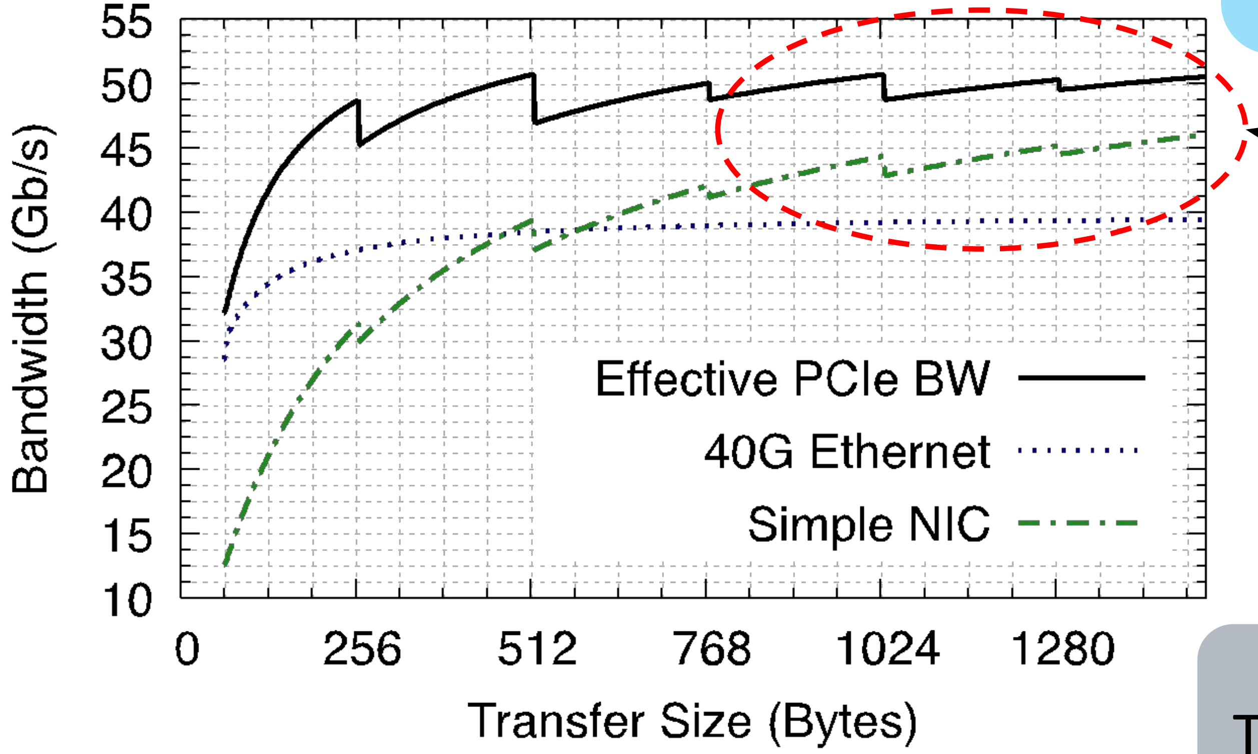


Why the sawtooth pattern?

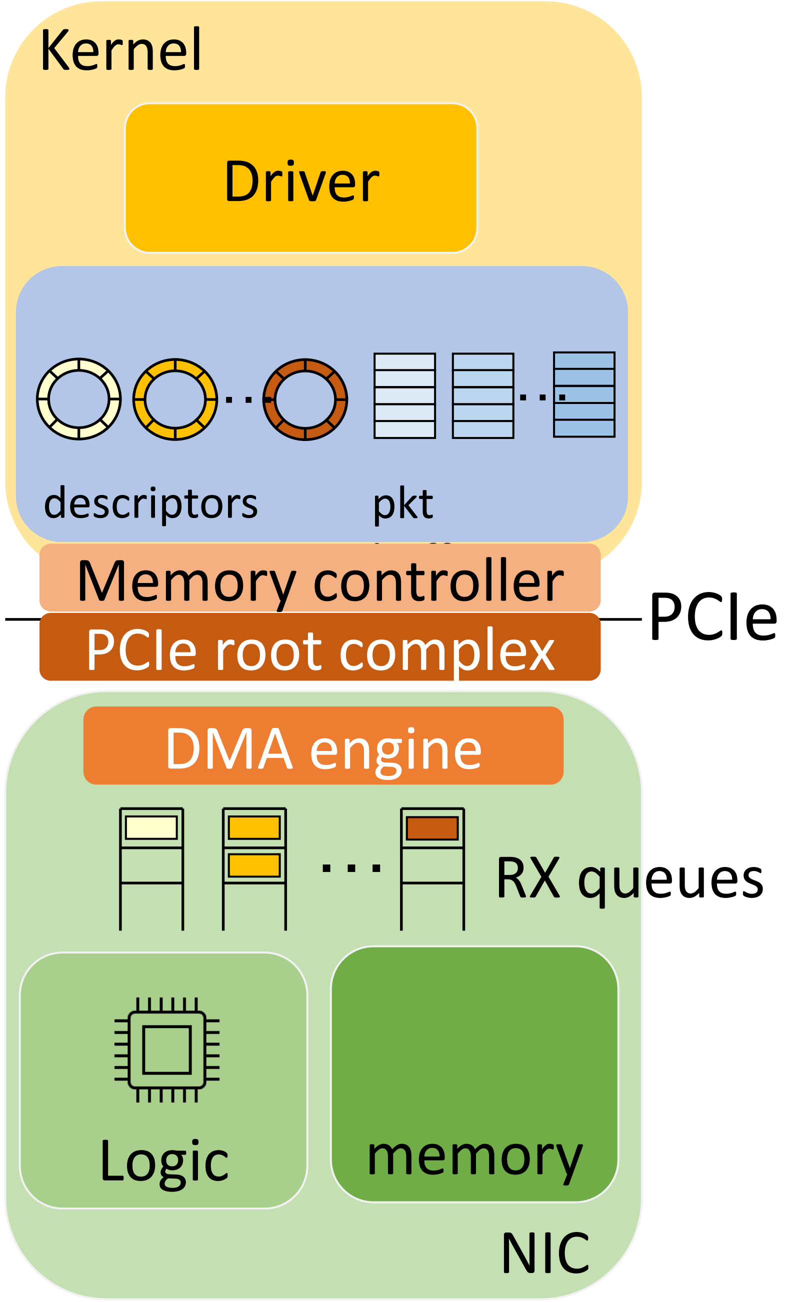
Because of the "packetized" nature of PCIe protocol

The impact of PCIe

With TSO/UFO/LRO you increase transfer size



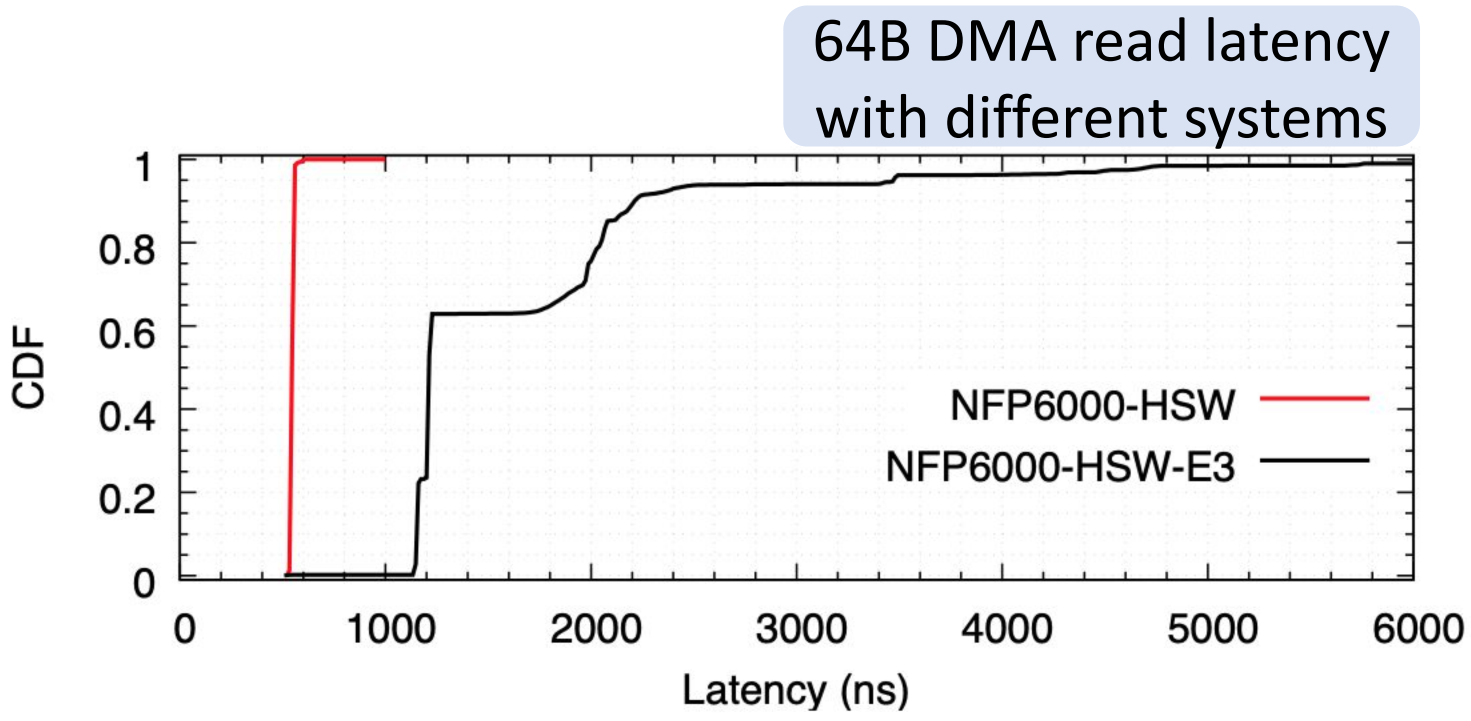
This why TSO/UFO/LRO are useful ☺

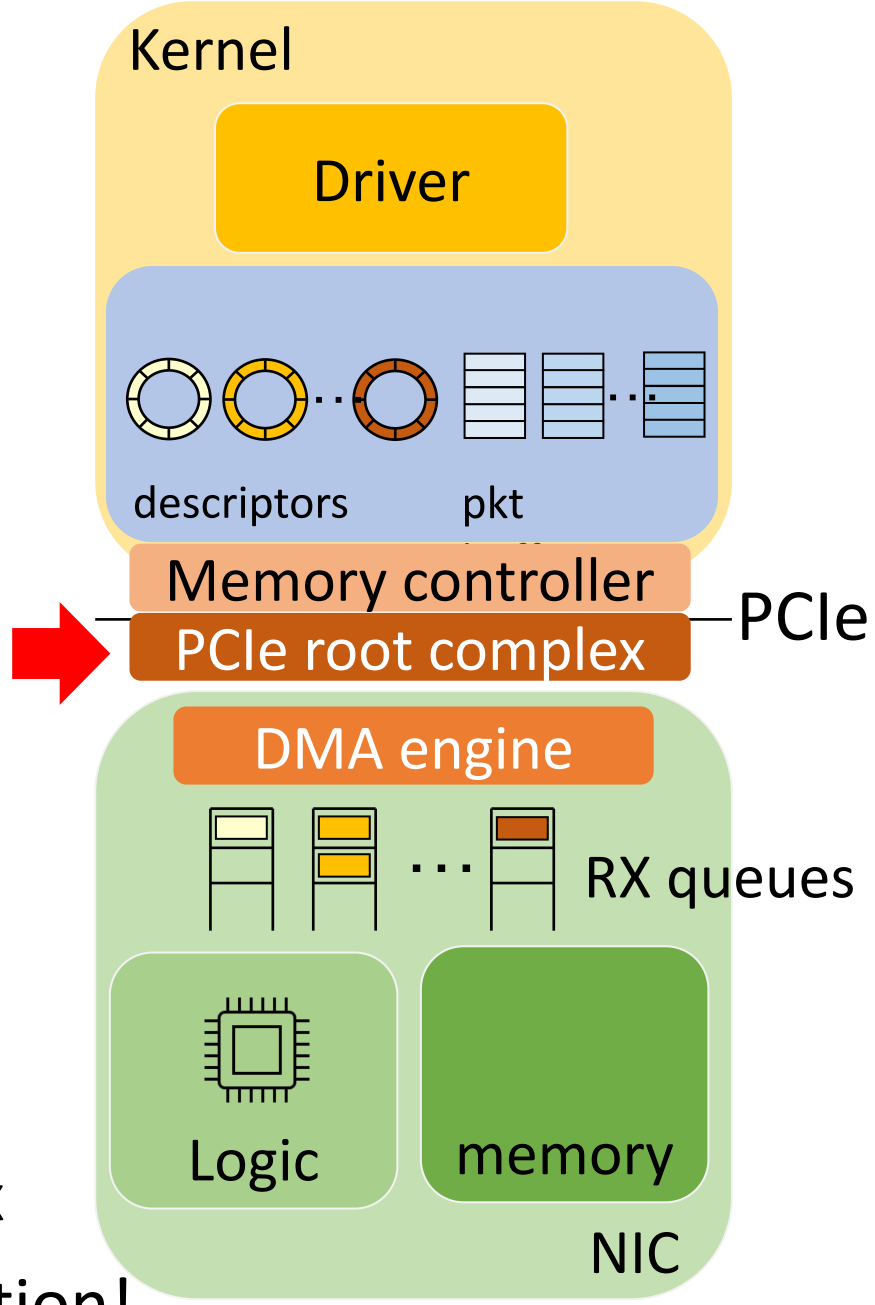


Why the sawtooth pattern?

Because of the "packetized" nature of PCIe protocol

The impact of PCIe





Xeon E5 has 547ns median latency and 1136ns max

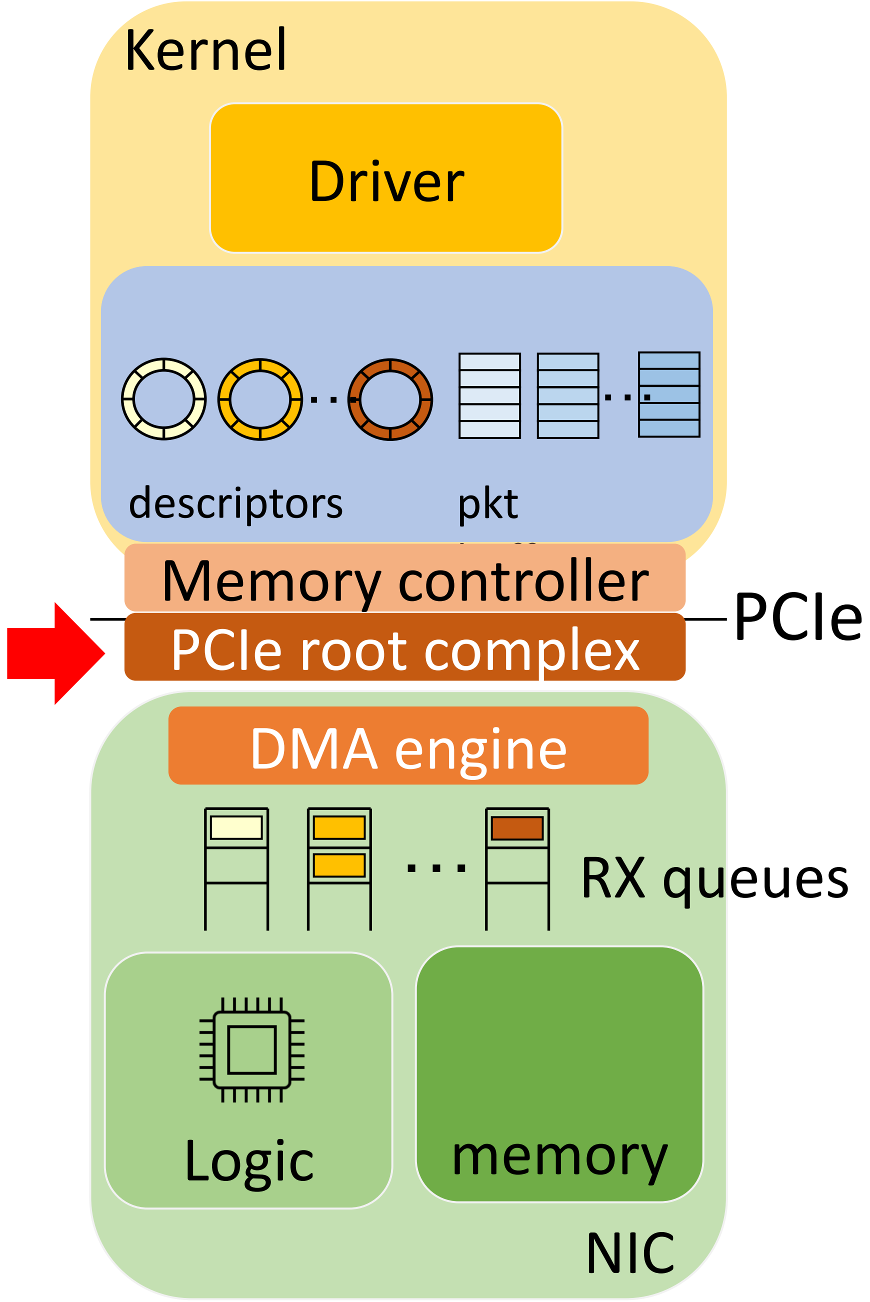
Xeon E3 has 1213ns median latency and 5.38ms max

This is dependent on PCIe root complex implementation!

PCIe is continuously evolving

time

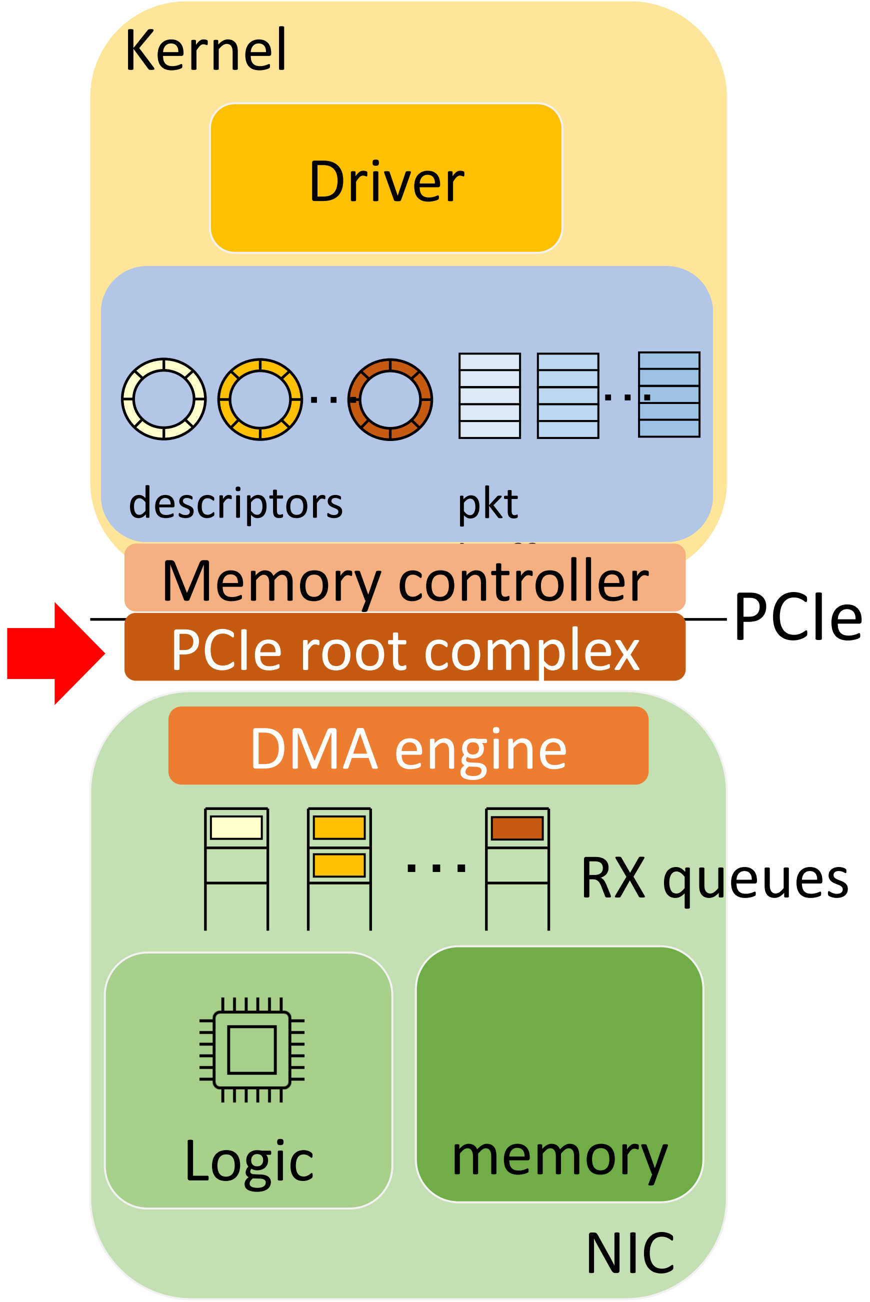
- 2003 — up to 4GBps
- 2007 — up to 8GBps
- 2010 — up to 16GBps
- 2017 — up to 32GBps
- 2019 — up to 63GBps
- 2022 — up to 121GBps



PCIe is continuously evolving

time

2003	●	up to 4GBps
2007	●	up to 8GBps
2010	●	up to 16GBps
2017	●	up to 32GBps
2019	●	up to 63GBps
2022	●	up to 121GBps
Today	●	PCIe is still evolving but also other standards are coming up



Compute eXpress Link

- Replacement for PCIe
  - Same physical layer, signalling and form factor
  - You can plug in a CXL or PCIe card, either/both will work
- Lower latency
  - Simplifies PCIe protocol to bring minimum latency down to around 200ns (PCIe can cost at least around 400ns)

NIC speeds: 200G today, 400G soon
At 400G, a 4kB packet is <u>80ns</u>
At 400G, a 64B packet is <u>1.25ns</u>

Compute eXpress Link

- Replacement for PCIe
  - Same physical layer, signalling and form factor
  - You can plug in a CXL or PCIe card, either/both will work
- Lower latency
  - Simplifies PCIe protocol to bring minimum latency down to around 200ns (PCIe can cost at least around 400ns)
- Cache coherent
  - A read from one device can be put in cache and accessed later without pain
  - This is because the device can invalidate the cache line

CXL Protocols

- CXL.io
  - PCIe-like config, I/O, interrupts. PCI-e backward compatible, non-coherent.
  - Command and control traffic and low-bandwidth management operations.

- CXL.cache
  - Cache coherent access from device to host memory.
  - Reduces data copying and latency for memory-intensive workloads.
  - Works with cache line granularity.

- CXL.mem
  - Cache coherent access from host to device memory.
  - Allows load/store operations to device memory regions.

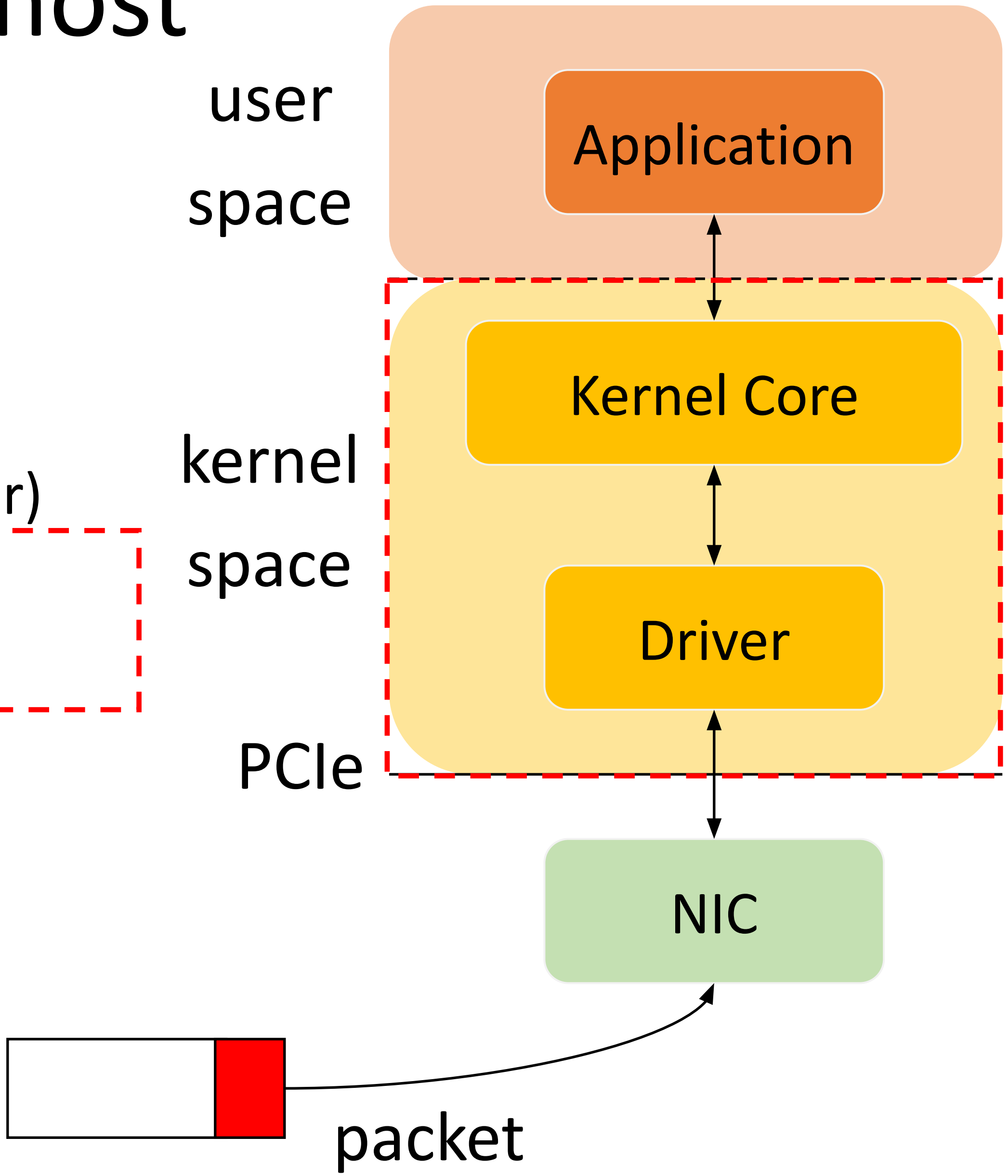
The software side

The NIC driver and the kernel

Life of a packet at the end-host

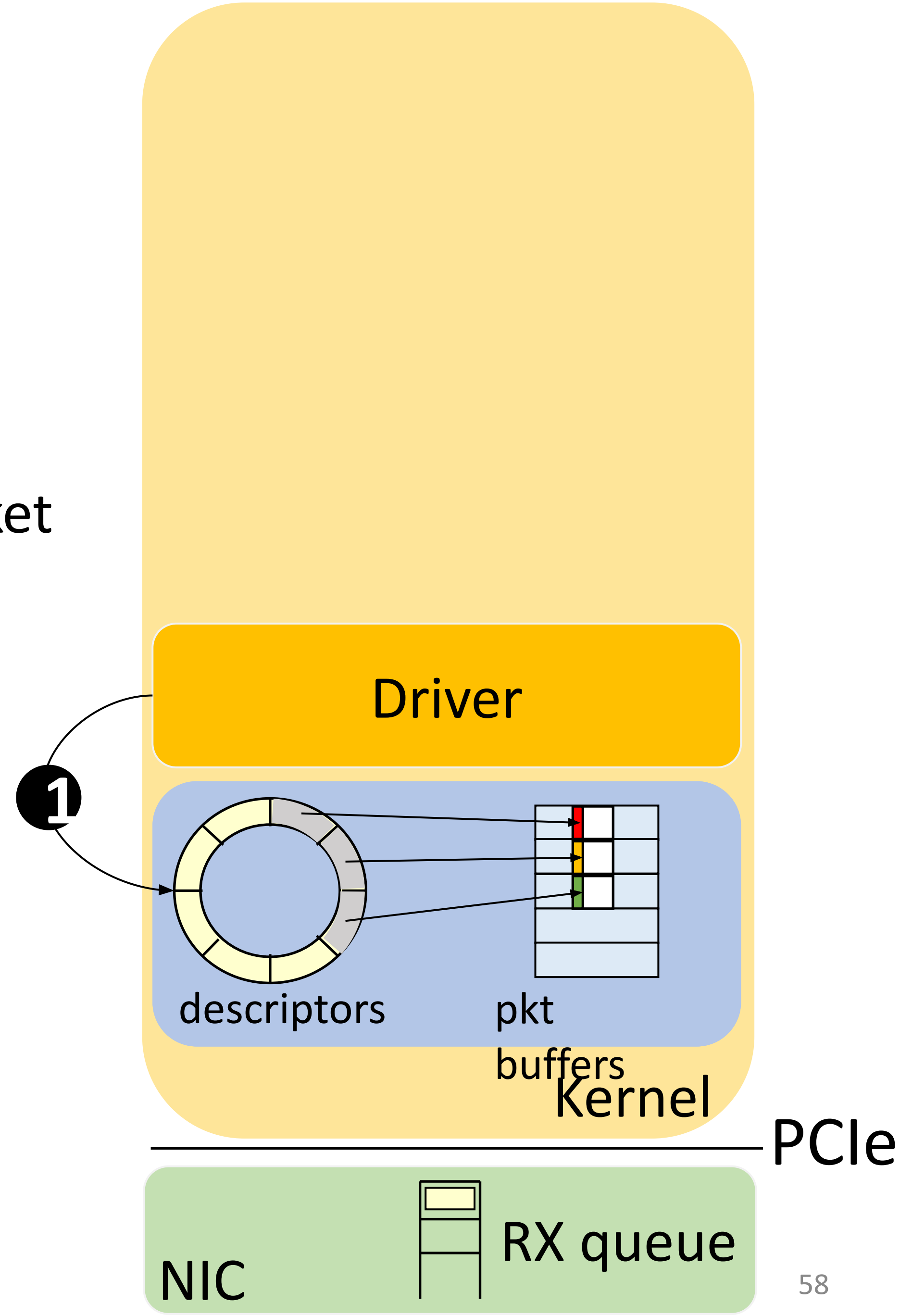
- From an high level perspective every packet has to cross:
  1. Network Interface Card (NIC)
  2. PCIe (interconnect between NIC and server)
  3. NIC driver
  4. Kernel

- …to finally reach the application in user space



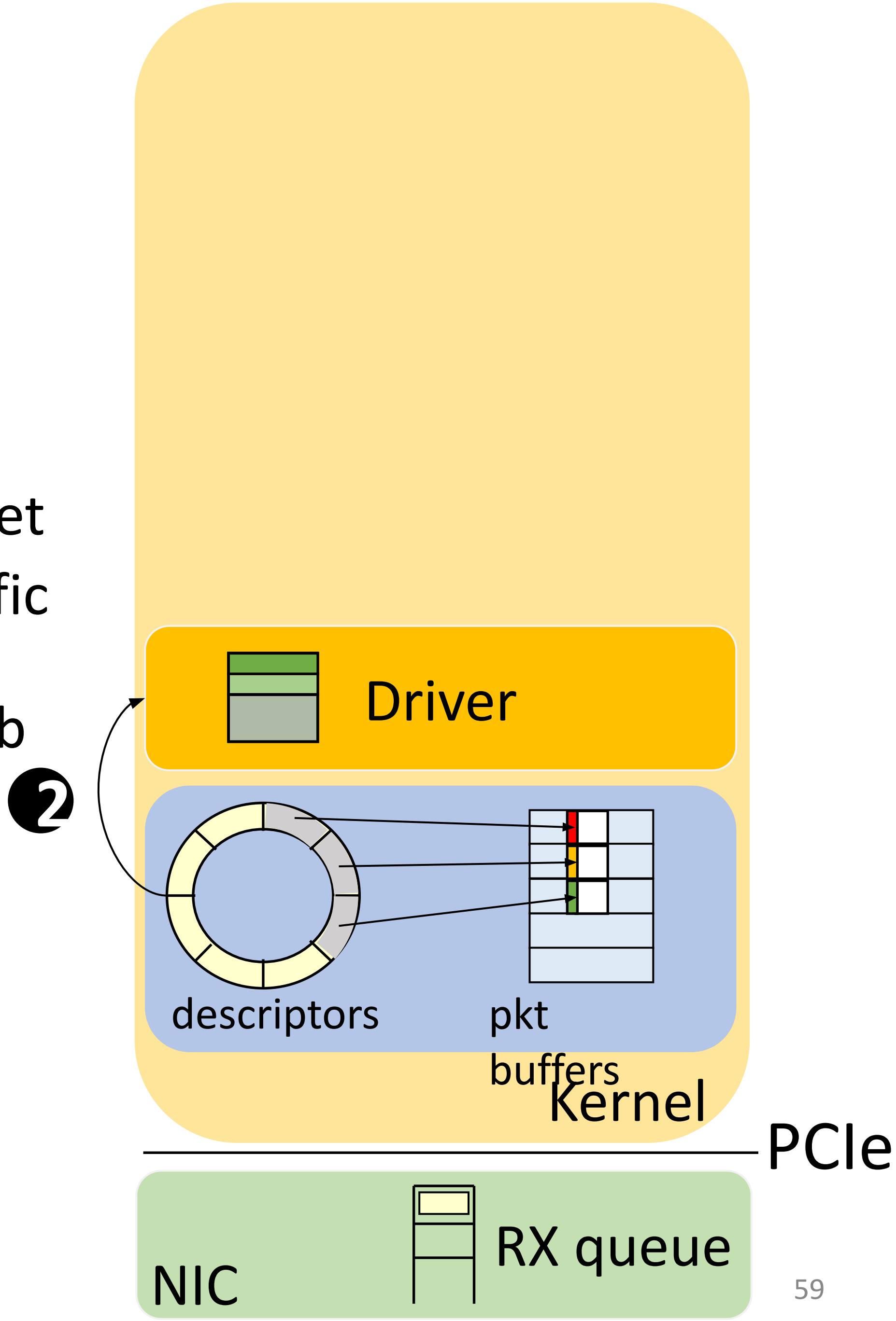
The NIC driver

- Regardless the NIC works in polling mode or interrupt-driven
  - The driver reads the descriptors ring to get a new packet



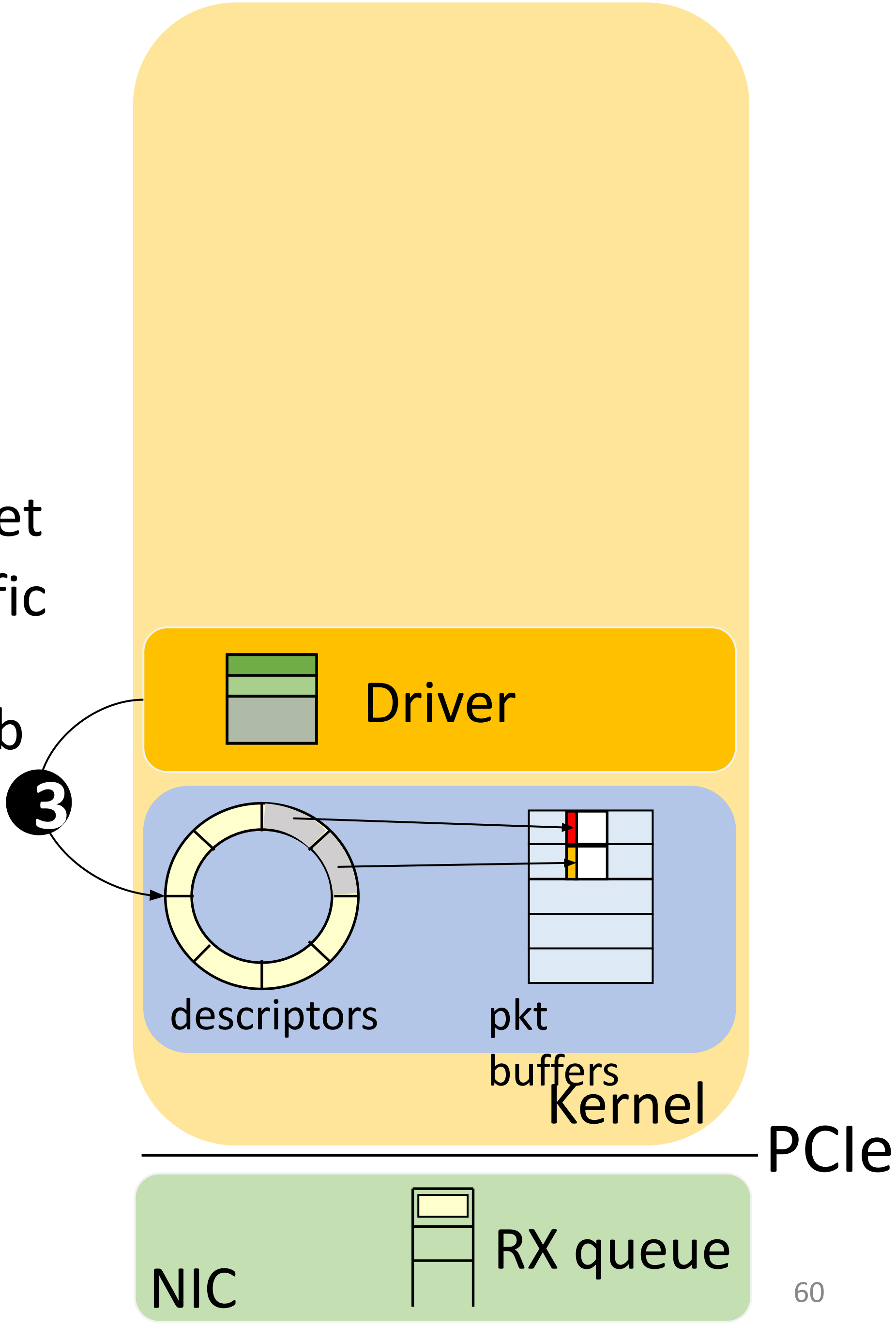
The NIC driver

- Regardless the NIC works in polling mode or interrupt-driven
  - The driver reads the descriptors ring to get a new packet
  - Fetches the packet from memory and allocates a specific data structure to handle it during its journey towards user-space: this is called **Socket Buffer** or sk_buff or skb



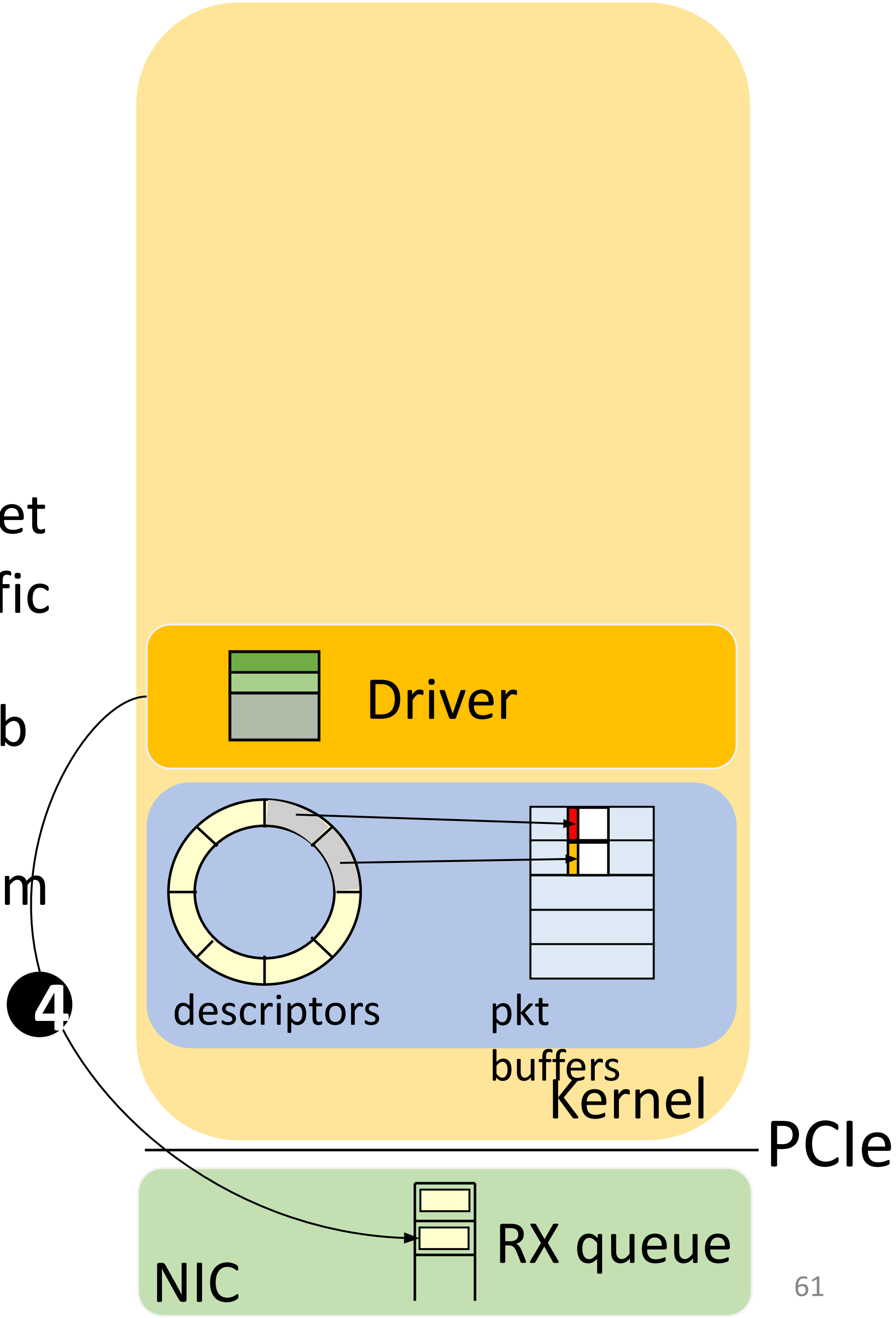
The NIC driver

- Regardless the NIC works in polling mode or interrupt-driven
  - The driver reads the descriptors ring to get a new packet
  - Fetches the packet from memory and allocates a specific data structure to handle it during its journey towards user-space: this is called **Socket Buffer** or sk_buff or skb
  - Clears the entry in the descriptor ring



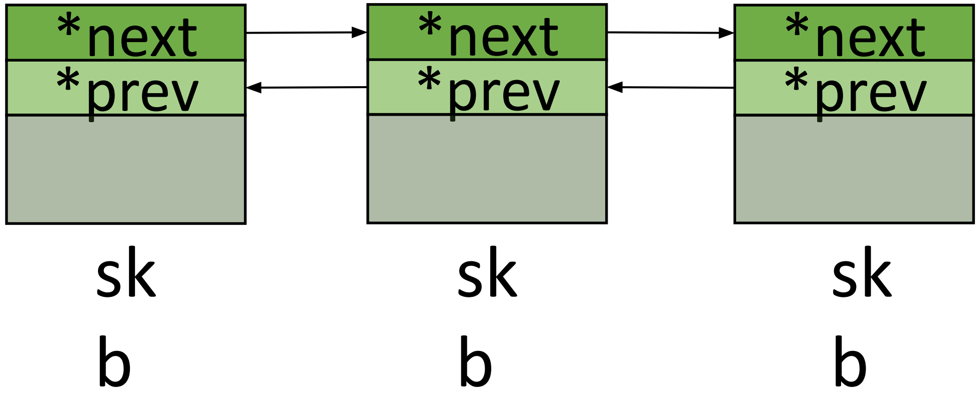
The NIC driver

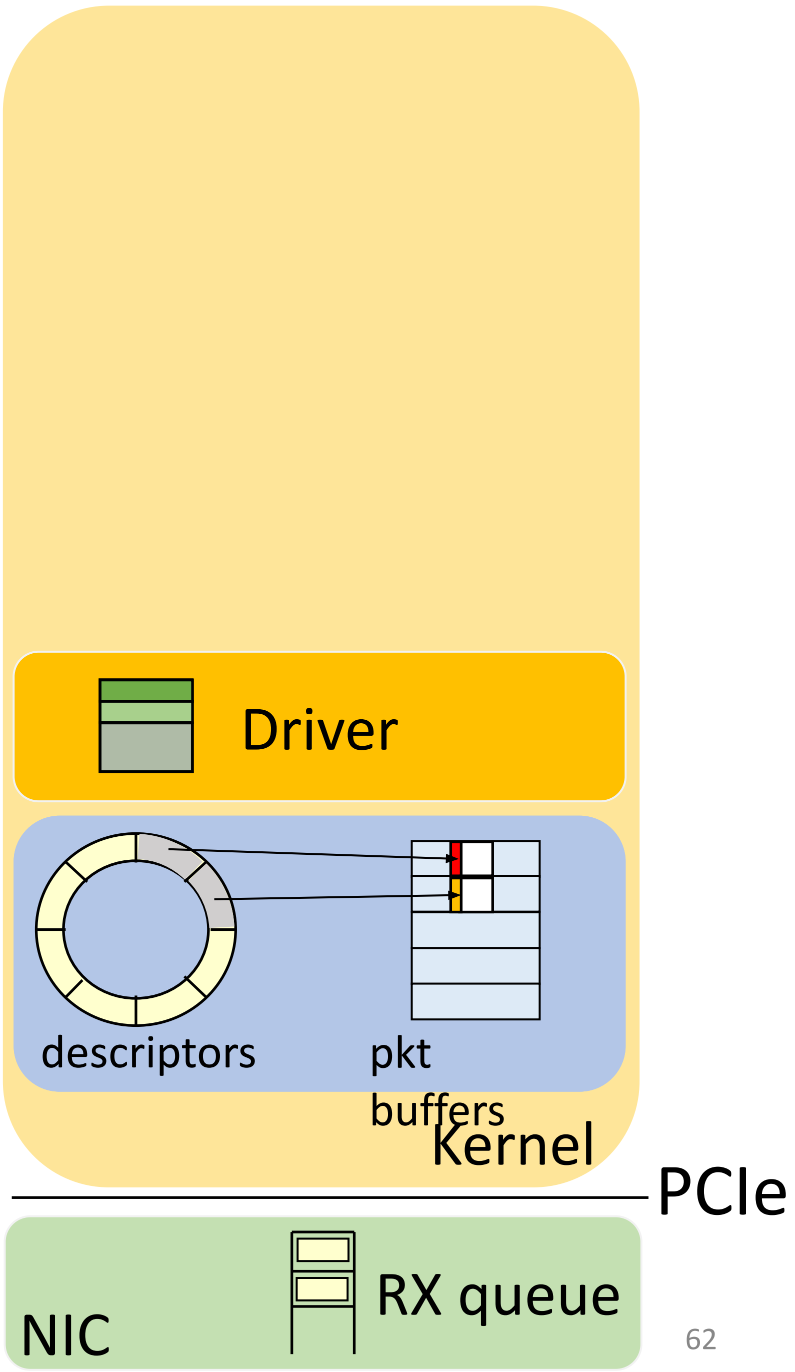
- Regardless the NIC works in polling mode or interrupt-driven
  - The driver reads the descriptors ring to get a new packet
  - Fetches the packet from memory and allocates a specific data structure to handle it during its journey towards user-space: this is called **Socket Buffer** or sk_buff or skb
  - Clears the entry in the descriptor ring
  - Writes back a new descriptor in the NIC's RX queue from a pool of free descriptors



The Socket Buffer

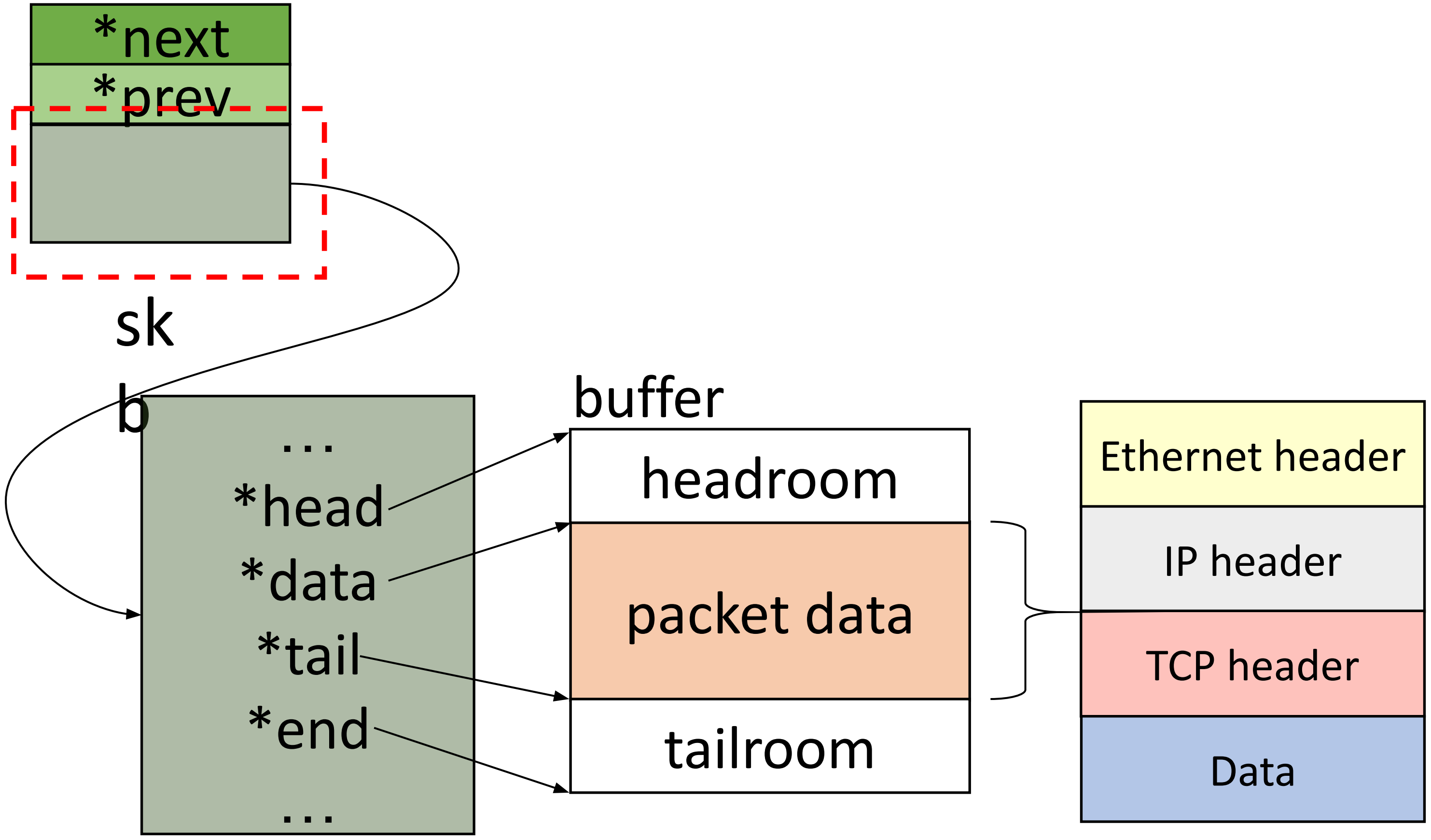
- Socket buffers are usually organised in circular lists to speed up certain operations (as we will see later)

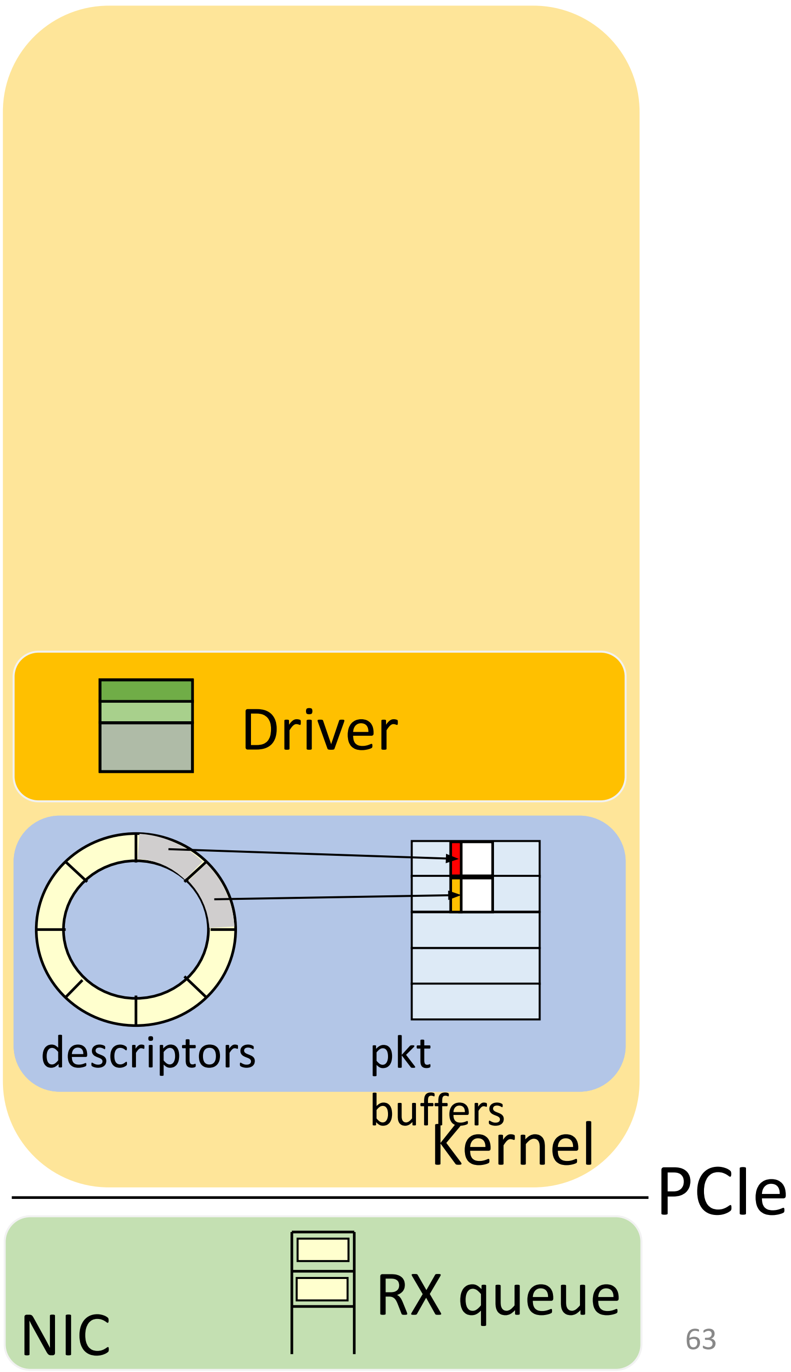




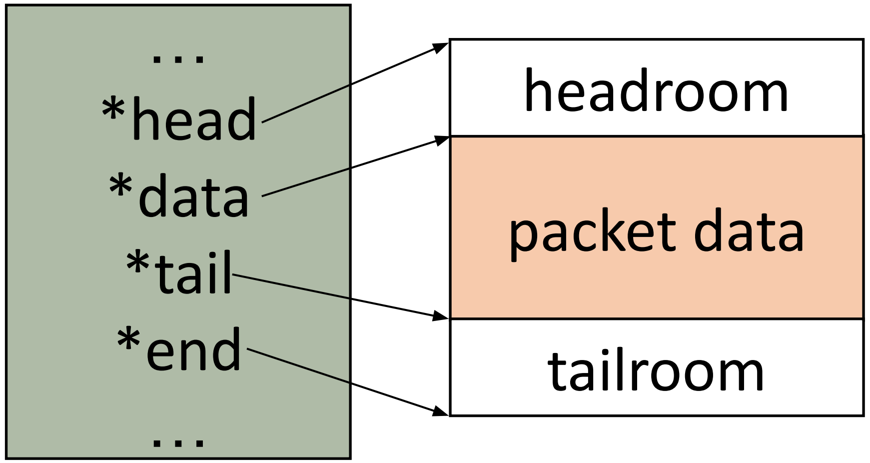
The Socket Buffer

- The skb data structure keeps reference to where the packet is stored in memory



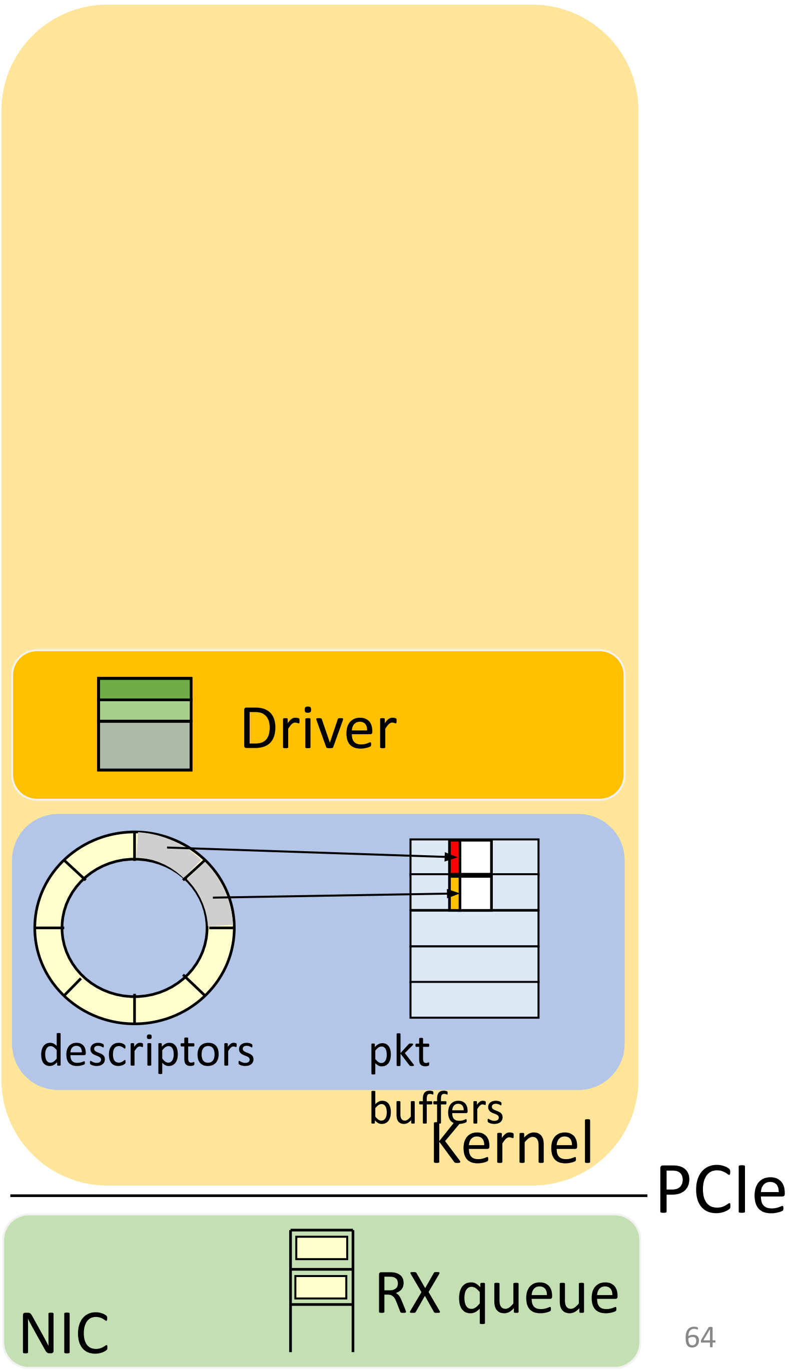


The Socket Buffer



- ***head*** and **end** points to the limits of the buffer
- ***data*** and ***tail*** delimit the packet data

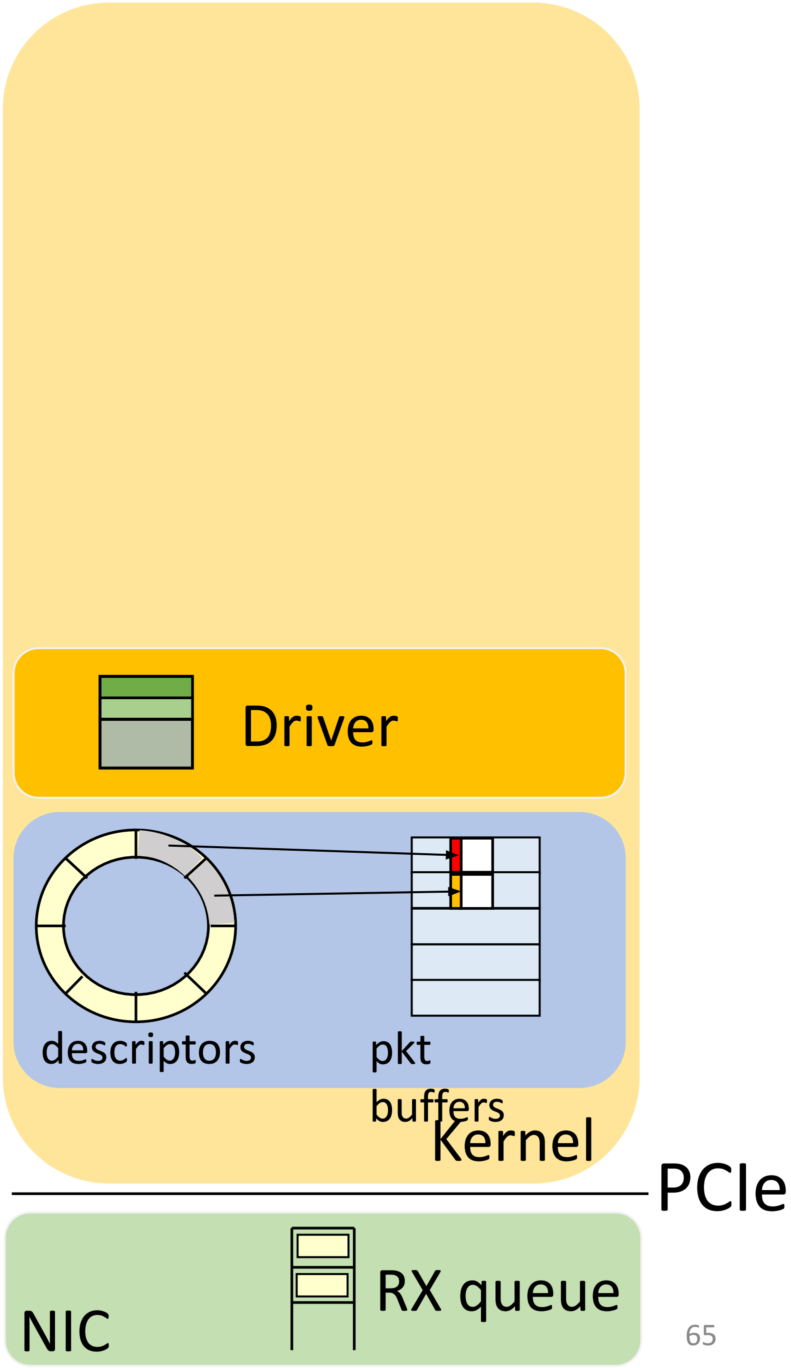
- ***data*** and ***tail*** can be moved (inside buffer) to create space for new headers without allocating a new structure



The Socket Buffer

- skb is more complex than that and can also store
  - Metadata (e.g., input interface, timestamp)
  - Kernel internal information
  - Value of the most useful fields to speed up processing

```
struct sk_buff {
        …
        struct net_device * dev; // The input or output device
        ktime_t tstamp; // The timestamp of reception
        __be16 protocol; //The L3 protocol of the packet
        __u16 transport_header; // Pointers to the varuous headers in the buffer
        __u16 network_header;
        …
}
```

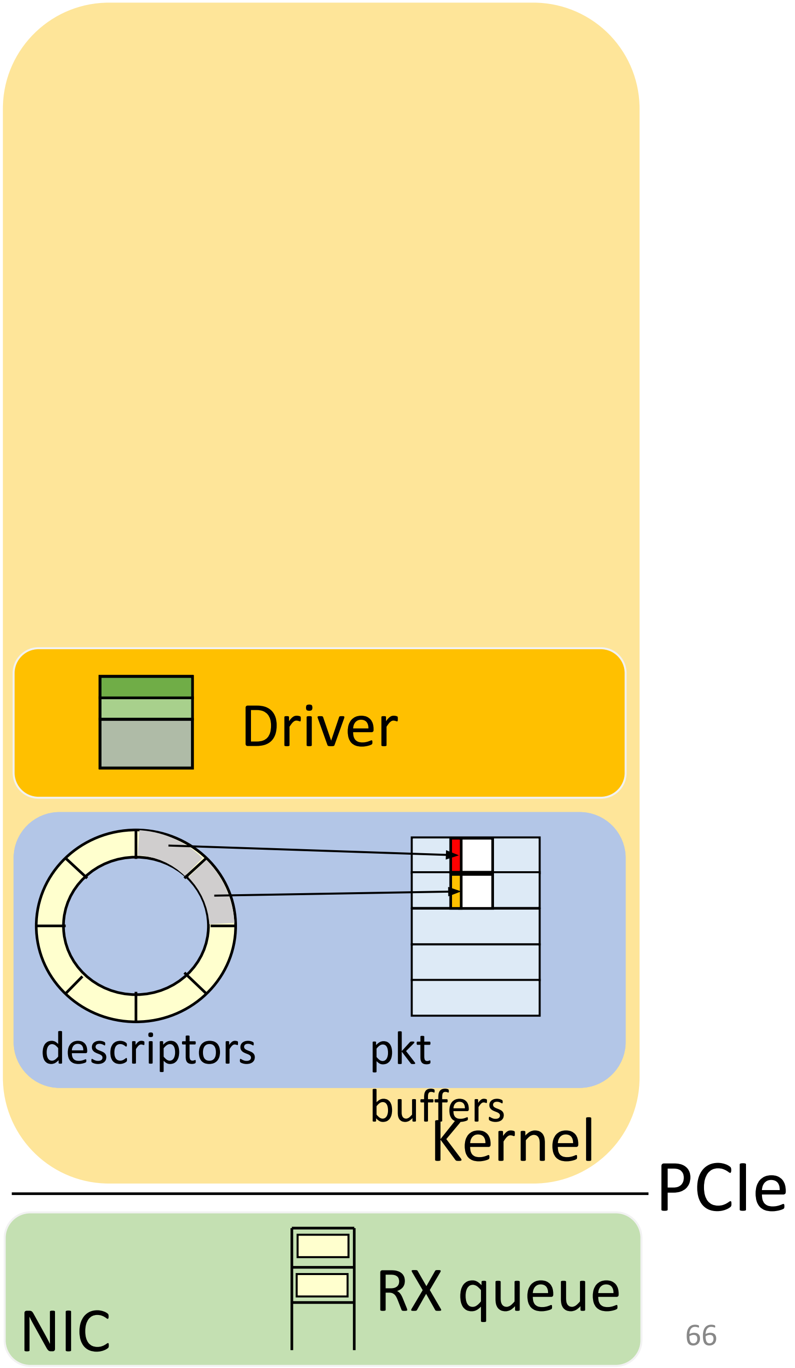


The Socket Buffer

- skb is more complex than that and can also store
  - Metadata (e.g., input interface, timestamp)
  - Kernel internal information
  - Value of the most useful fields to speed up processing

```
struct sk_buff {
        …
        struct net_device * dev; // The input or output device
        ktime_t tstamp; // The timestamp of reception
        __be16 protocol; //The L3 protocol of the packet
        __u16 transport_header; // Pointers to the varuous headers in the buffer
        __u16 network_header;
        …
}
```

..and many more!



The Socket Buffer

```
* @ip_summed: Driver fed us an IP checksum
* @nohdr: Payload reference only, must not modify header
* @pkt_type: Packet class
* @fclone: skbuff clone status
* @ipvs_property: skbuff is owned by ipvs
* @inner_protocol_type: whether the inner protocol is
*    ENCAP_TYPE_ETHER or ENCAP_TYPE_IPPROTO
* @remcsum_offload: remote checksum offload is enabled
* @offload_fwd_mark: Packet was L2-forwarded in hardware
* @offload_l3_fwd_mark: Packet was L3-forwarded in hardware
* @tc_skip_classify: do not classify packet. set by IFB device
* @tc_at_ingress: used within tc_classify to distinguish in/egress
* @redirected: packet was redirected by packet classifier
* @from_ingress: packet was redirected from the ingress path
* @nf_skip_egress: packet shall skip nf egress - see netfilter_netdev.h
* @peeked: this packet has been seen already, so stats have been
*    done for it, don't do them again
* @nf_trace: netfilter packet trace flag
* @protocol: Packet protocol from driver
* @destructor: Destruct function
* @tcp_tsorted_anchor: list structure for TCP (tp->tsorted_sent_queue)
* @_sk_redir: socket redirection information for skmsg
* @_nfct: Associated connection, if any (with nfctinfo bits)
* @nf_bridge: Saved data about a bridged frame - see br_netfilter.c
* @skb_iif: ifindex of device we arrived on
* @tc_index: Traffic control index
* @hash: the packet hash
* @queue_mapping: Queue mapping for multiqueue devices
* @head_frag: skb was allocated from page fragments,
*    not allocated by kmalloc() or vmalloc().
* @pfmemalloc: skbuff was allocated from PFMEMALLOC reserves
* @pp_recycle: mark the packet for recycling instead of freeing
*    (implies page_pool support on driver)
* @active_extensions: active extensions (skb_ext_id types)
* @ndisc_nodetype: router type (from link layer)
* @ooo_okay: allow the mapping of a socket to a queue to be changed
* @l4_hash: indicate hash is a canonical 4-tuple hash over transport
*    ports.
* @sw_hash: indicates hash was computed in software stack
* @wifi_acked_valid: wifi_acked was set
* @wifi_acked: whether frame was acked on wifi or not
* @no_fcs:    Request NIC to treat last 4 bytes as Ethernet FCS
* @encapsulation: indicates the inner headers in the skbuff are valid
* @encap_hdr_csum: software checksum is needed
* @csum_valid: checksum is already valid
* @csum_not_inet: use CRC32c to resolve CHECKSUM_PARTIAL
* @csum_complete_sw: checksum was completed by software
* @csum_level: indicates the number of consecutive checksums
*    found in the packet minus one that have been verified as
*    CHECKSUM_UNNECESSARY (max 3)
```

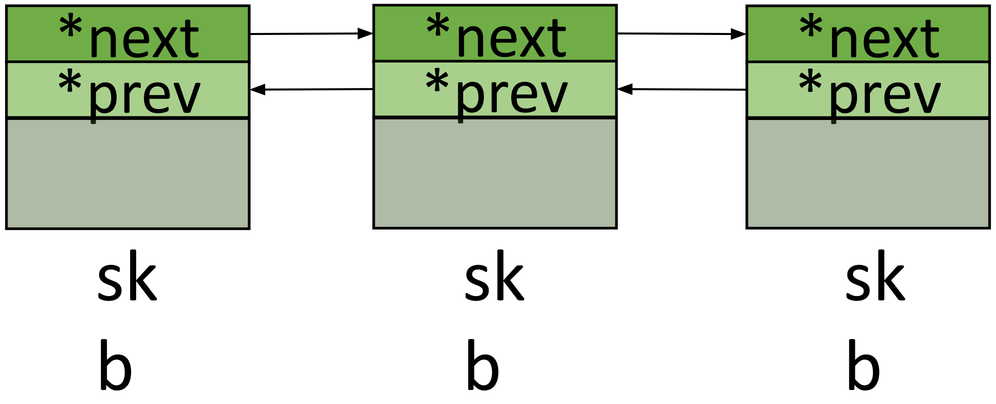
The Socket Buffer

```
* @ip_summed: Driver fed us an IP checksum
* @nohdr: Payload reference only, must not modify header
* @pkt_type: Packet class
* @fclone: skbuff clone status
* @ipvs_property: skbuff is owned by ipvs
* @inner_protocol_type: whether the inner protocol is
*   ENCAP_TYPE_ETHER or ENCAP_TYPE_IPPROTO
* @remcsum_offload: remote checksum offload is enabled
* @offload_fwd_mark: Packet was L2-forwarded in hardware
* @offload_l3_fwd_mark: Packet was L3-forwarded in hardware
* @tc_skip_classify: do not classify packet. set by IFB device
* @tc_at_ingress: used within tc_classify to distinguish in/egress
* @redirected: packet was redirected by packet classifier
* @from_ingress: packet was redirected from the ingress path
* @nf_skip_egress: packet shall skip nf egress - see netfilter_netdev.h
* @peeked: this packet has been seen already, so stats have been
*   done for it, don't do them again
* @nf_trace: netfilter packet trace flag
* @protocol: Packet protocol from driver
* @destructor: Destruct function
* @tcp_tsorted_anchor: list structure for TCP (tp->tsorted_sent_queue)
* @_sk_redir: socket redirection information for skmsg
* @_nfct: Associated connection, if any (with nfctinfo bits)
* @nf_bridge: Saved data about a bridged frame - see br_netfilter.c
* @skb_iif: ifindex of device we arrived on
* @tc_index: Traffic control index
* @hash: the packet hash
* @queue_mapping: Queue mapping for multiqueue devices
* @head_frag: skb was allocated from page fragments,
*   not allocated by kmalloc() or vmalloc().
* @pfmemalloc: skbuff was allocated from PFMEMALLOC reserves
* @pp_recycle: mark the packet for recycling instead of freeing
*   (implies page_pool support on driver)
* @active_extensions: active extensions (skb_ext_id types)
* @ndisc_nodetype: router type (from link layer)
* @ooo_okay: allow the mapping of a socket to a queue to be changed
* @l4_hash: indicate hash is a canonical 4-tuple hash over transport
*   ports.
* @sw_hash: indicates hash was computed in software stack
* @wifi_acked_valid: wifi_acked was set
* @wifi_acked: whether frame was acked on wifi or not
* @no_fcs:   Request NIC to treat last 4 bytes as Ethernet FCS
* @encapsulation: indicates the inner headers in the skbuff are valid
* @encap_hdr_csum: software checksum is needed
* @csum_valid: checksum is already valid
* @csum_not_inet: use CRC32c to resolve CHECKSUM_PARTIAL
* @csum_complete_sw: checksum was completed by software
* @csum_level: indicates the number of consecutive checksums
*   found in the packet minus one that have been verified as
*   CHECKSUM_UNNECESSARY (max 3)
```

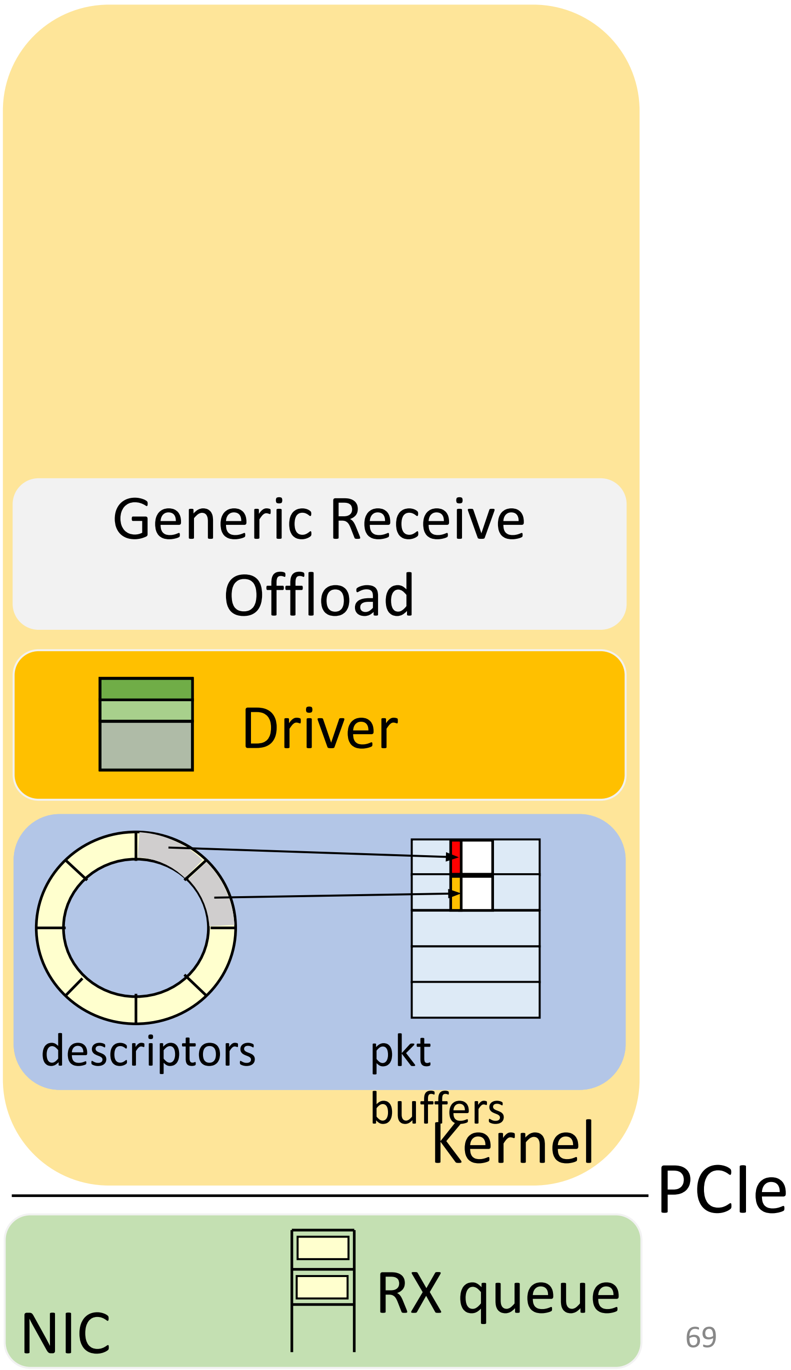
sizeof(sk_buff) ≈ 400 B

The network stack: GRO

- The Generic Receive Offload (GRO) module attempts to reduce the number of skbs by merging them (having them organized circularly helps!)

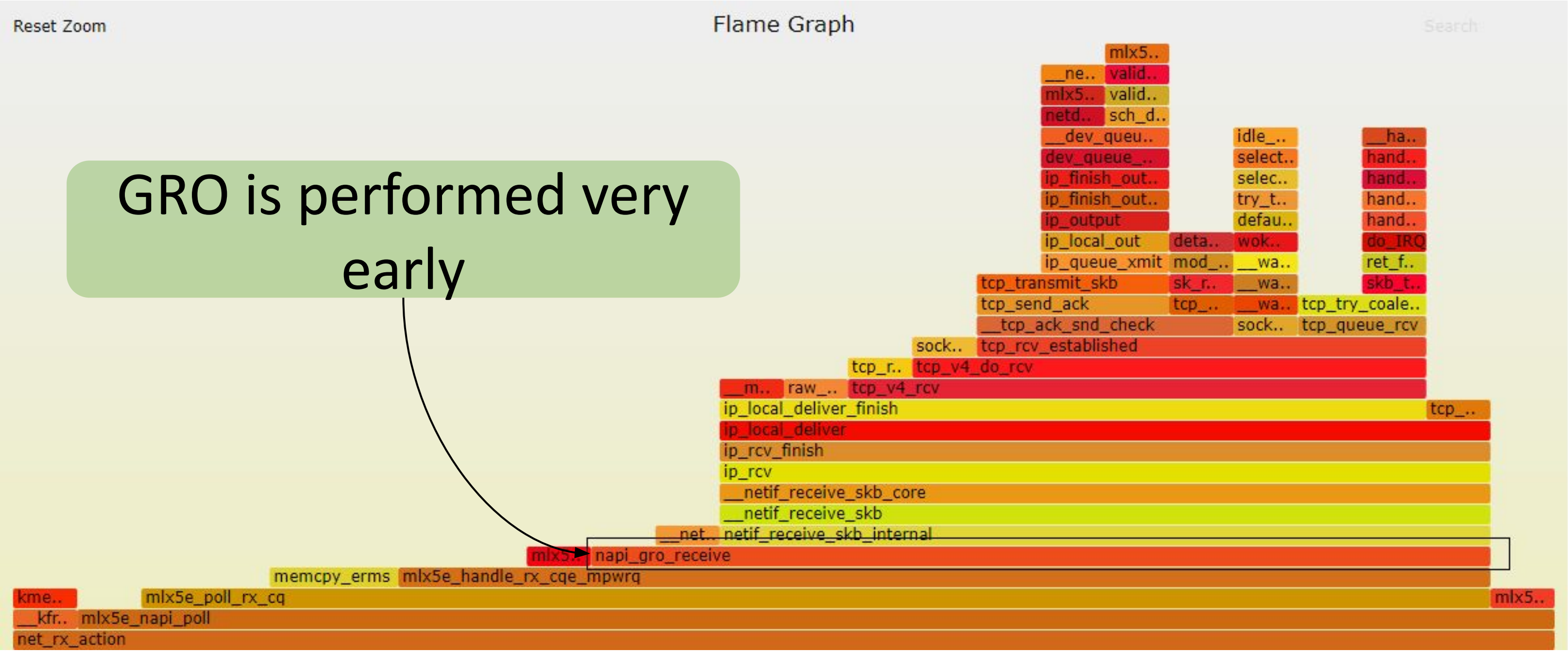


- This operation is the software counterpart of the Large Received Offload (LRO), which is performed by the NIC.



GRO

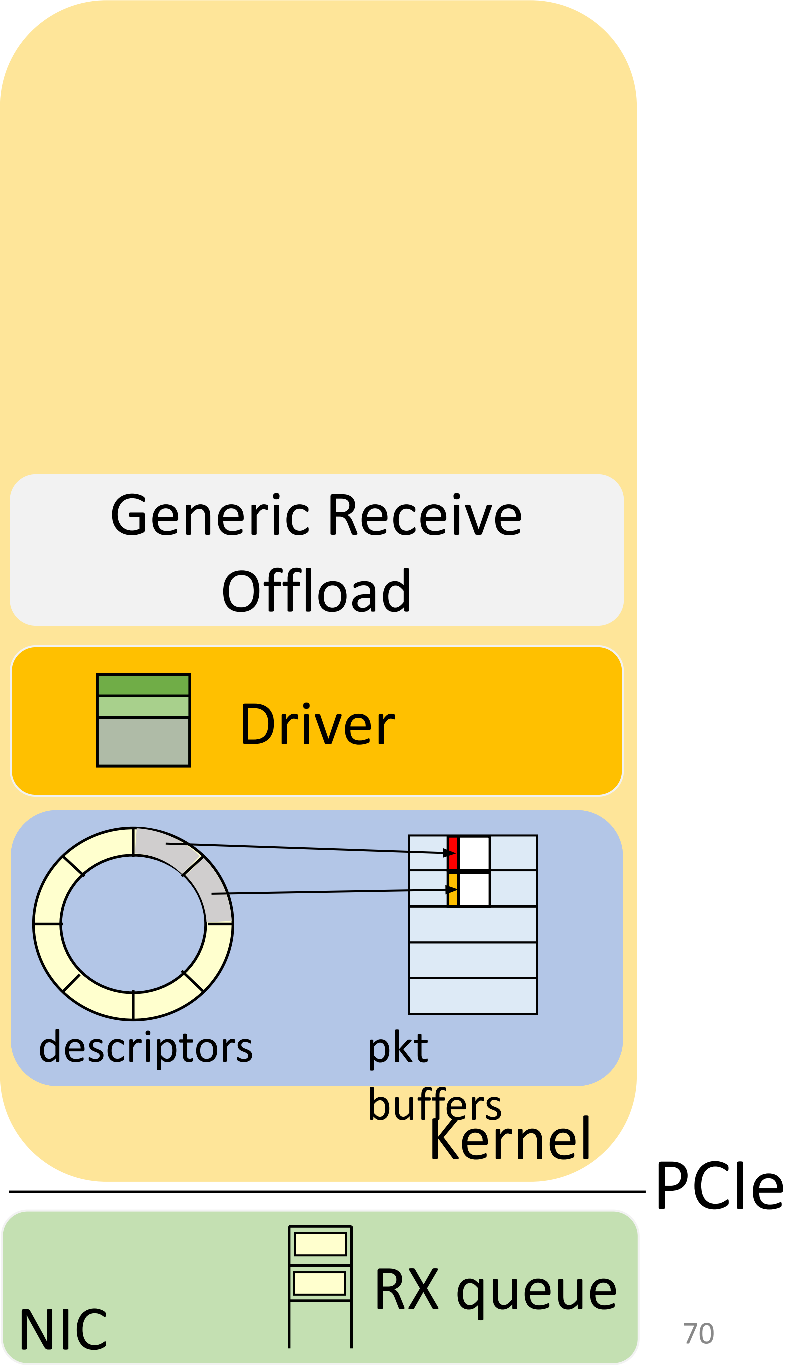
Mellanox ConnectX-4 LX



GRO is performed very early

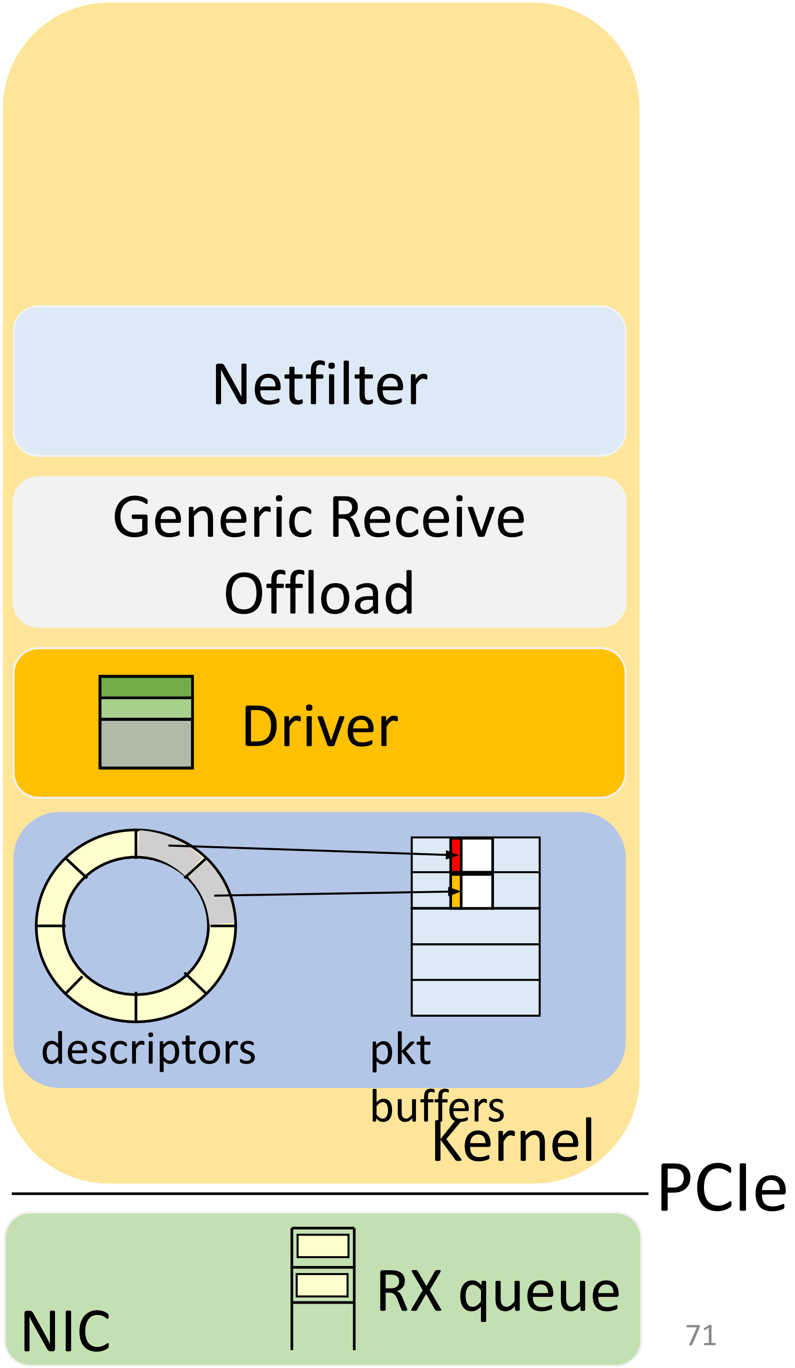
https://stackoverflow.com/questions/47332232/why-is-gro-more-efficient

This means that all the next functions in the receive stack do much less processing (you act on a single skb for multiple packets)



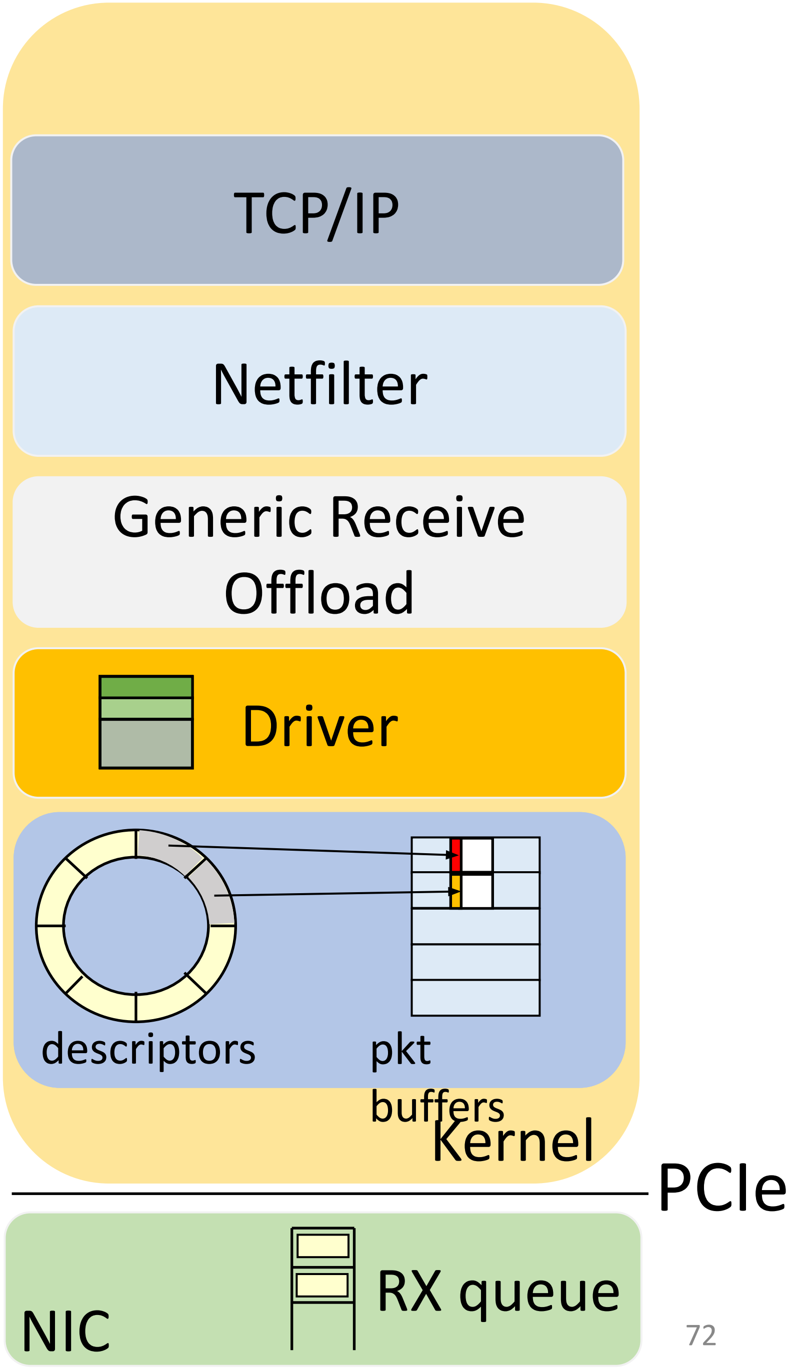
The network stack: Netfilter

- Netfilter is a framework that offers various functions and operations for packet filtering, network address translation (NAT), port translation and connection tracking
- The IPtables Linux firewall leverages Netfilter to implement its policies



The network stack: TCP/IP

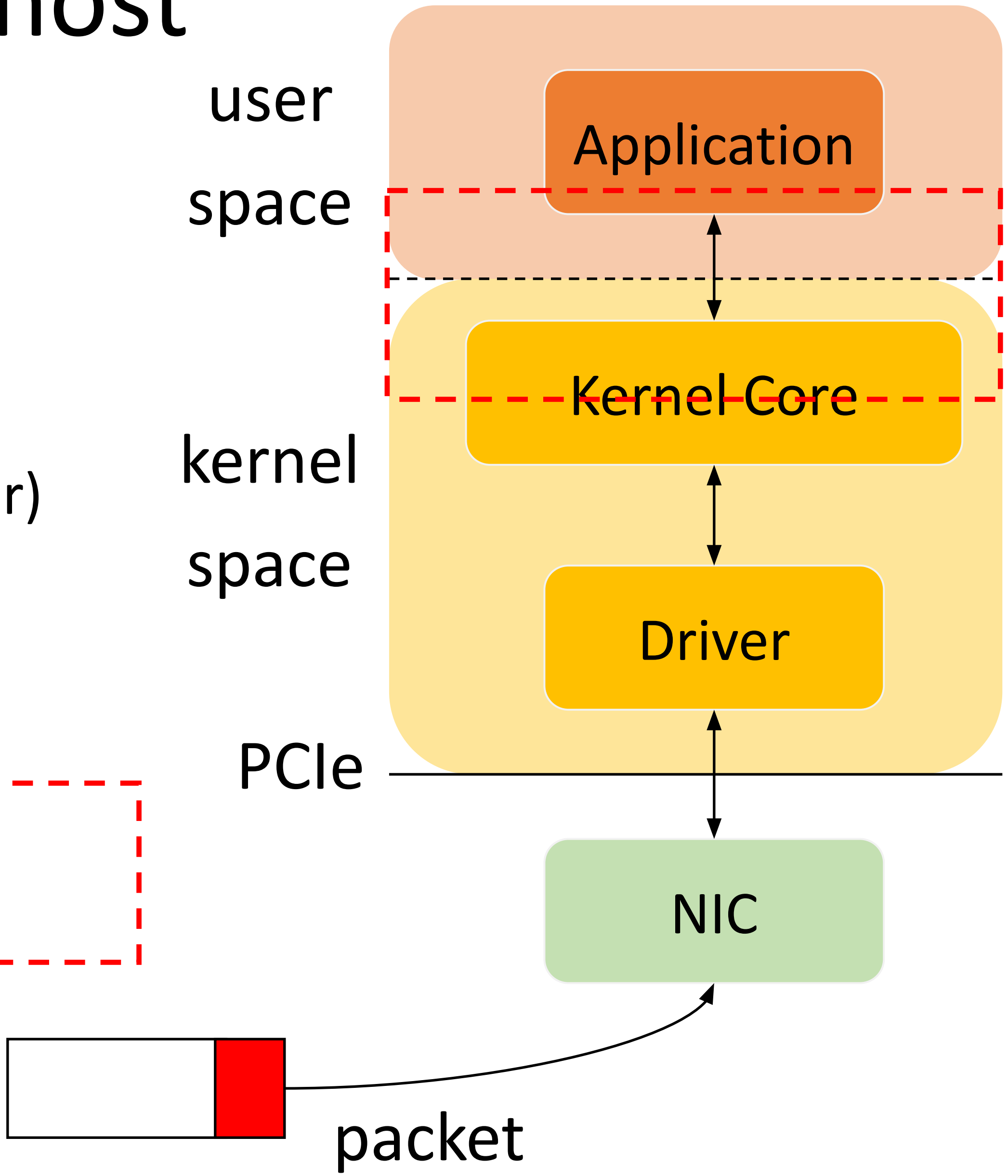
- Here all the processing related to the transport layer are performed.
- For TCP packets, this means, for example:
  - Generating ACKs
  - Updating the congestion window
  - Check TCP checksum



Life of a packet at the end-host

- From an high level perspective every packet has to cross:
  1. Network Interface Card (NIC)
  2. PCIe (interconnect between NIC and server)
  3. NIC driver
  4. Kernel

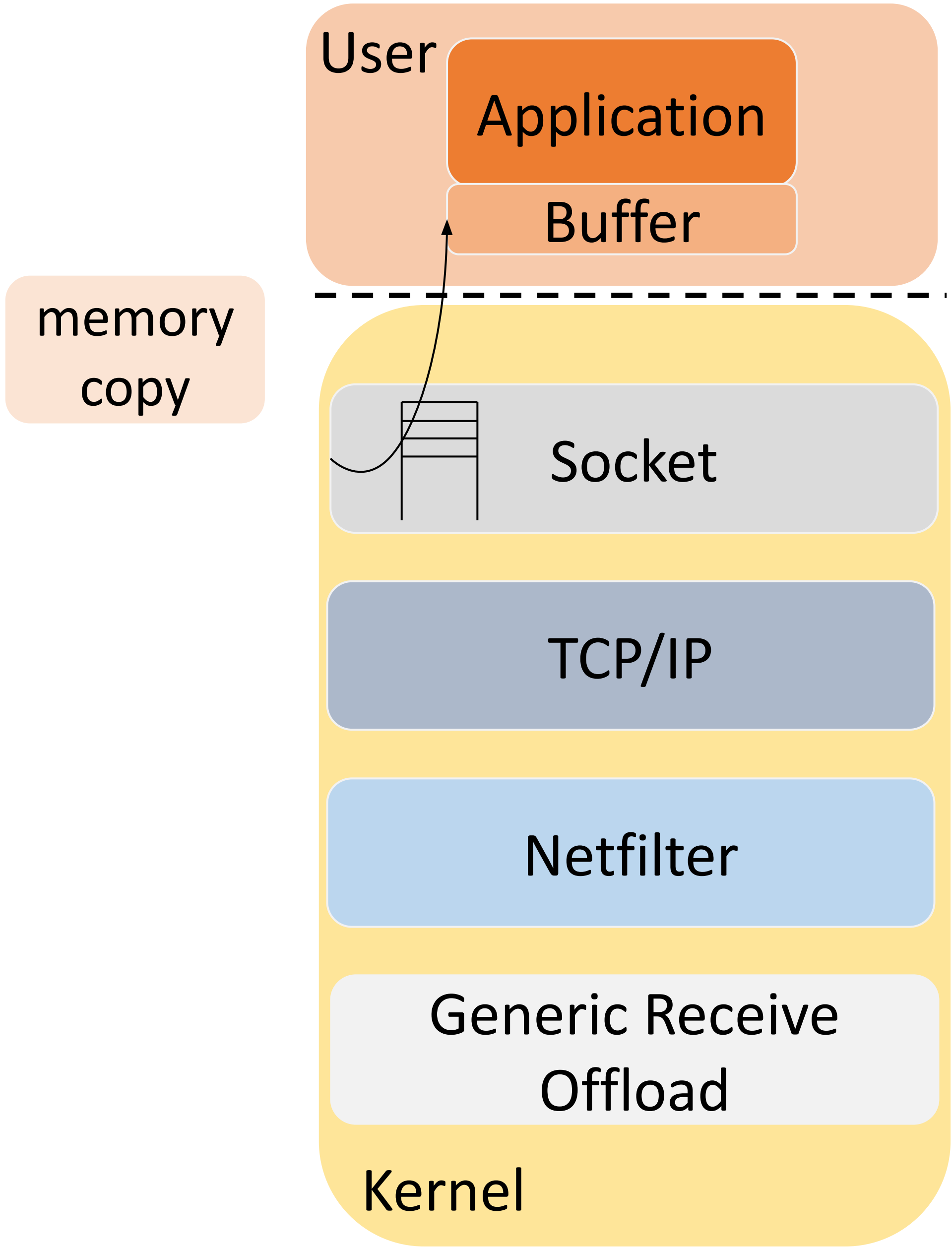
- …to finally reach the application in user space



Kernel to User Space

- After the TCP/IP processing, all in-order skbs are appended to the application socket's receive queue

- The application performs data copy of the payload in the skbs in the socket receive queue to the userspace buffer

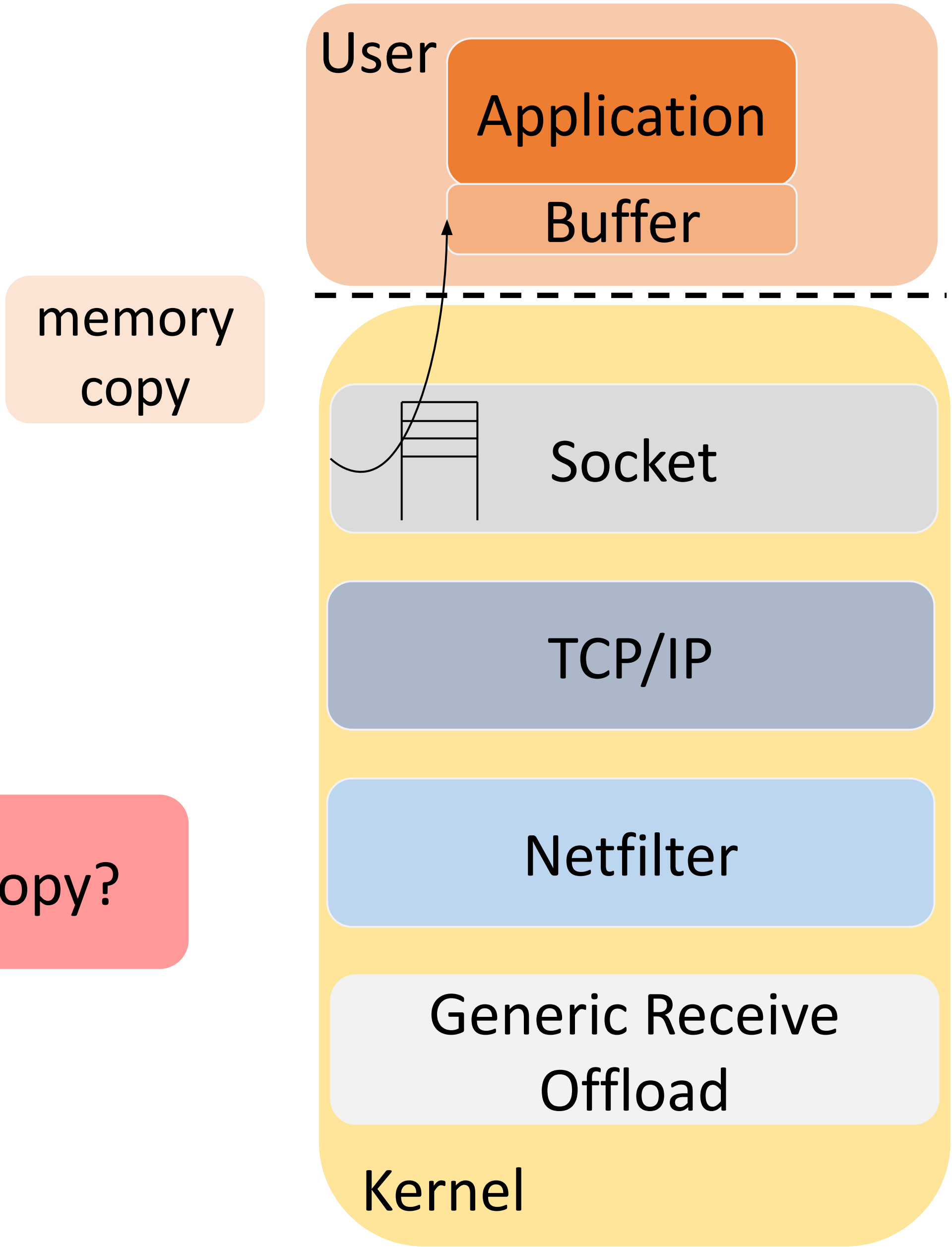


Kernel to User Space

- After the TCP/IP processing, all in-order skbs are appended to the application socket's receive queue

- The application performs data copy of the payload in the skbs in the socket receive queue to the userspace buffer

Why a data copy?



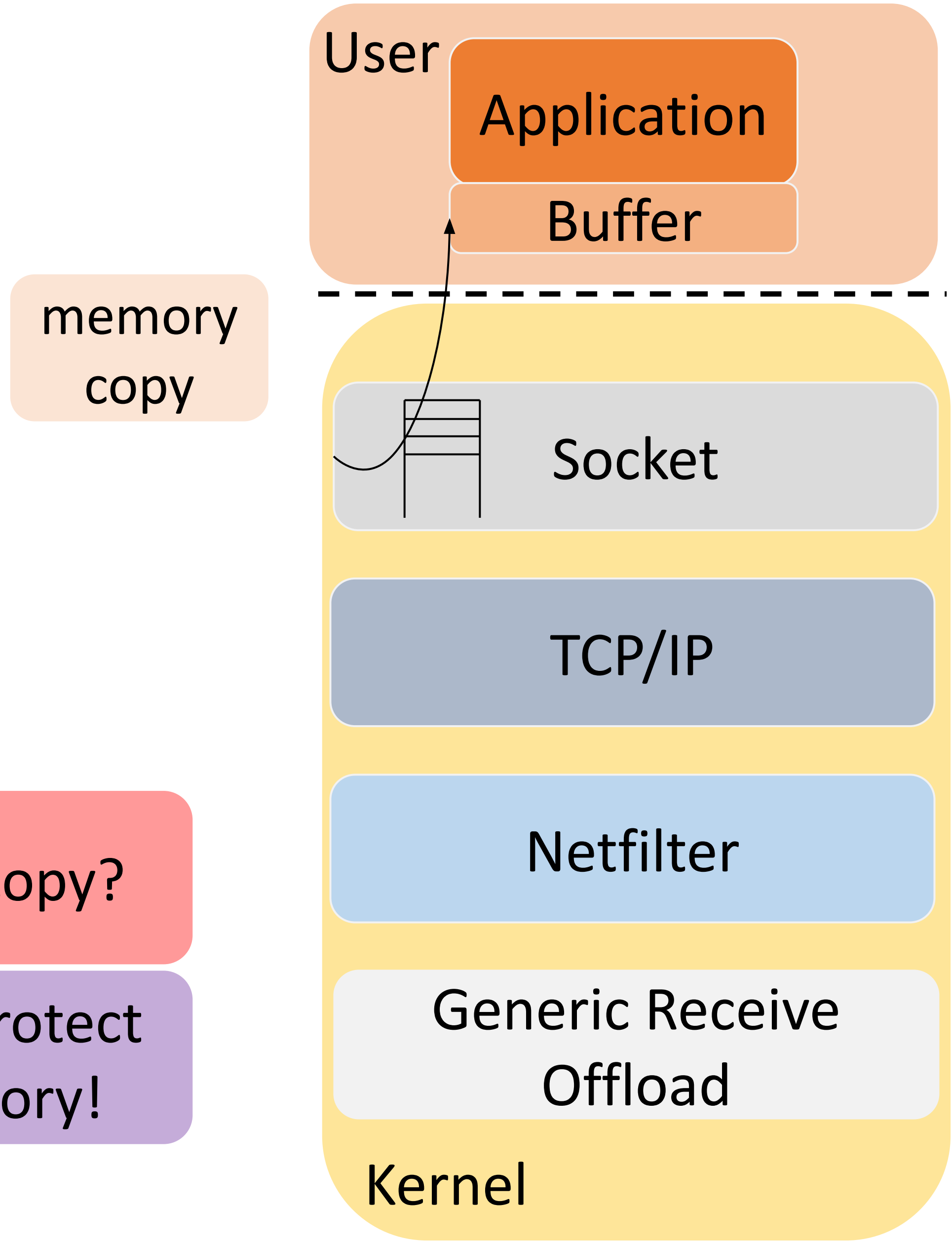
Kernel to User Space

- After the TCP/IP processing, all in-order skbs are appended to the application socket's receive queue

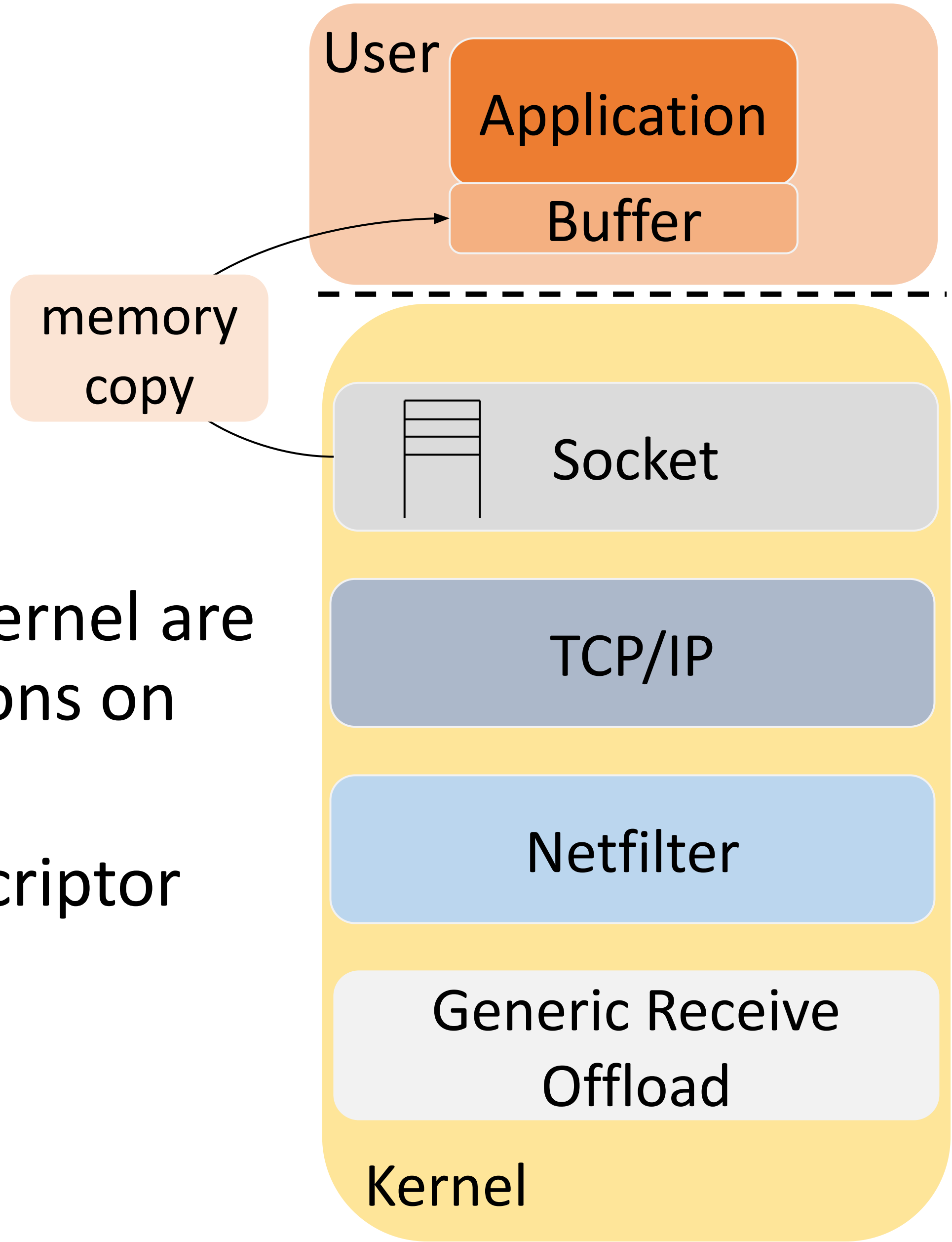
- The application performs data copy of the payload in the skbs in the socket receive queue to the userspace buffer

Why a data copy?

we need to protect kernel memory!



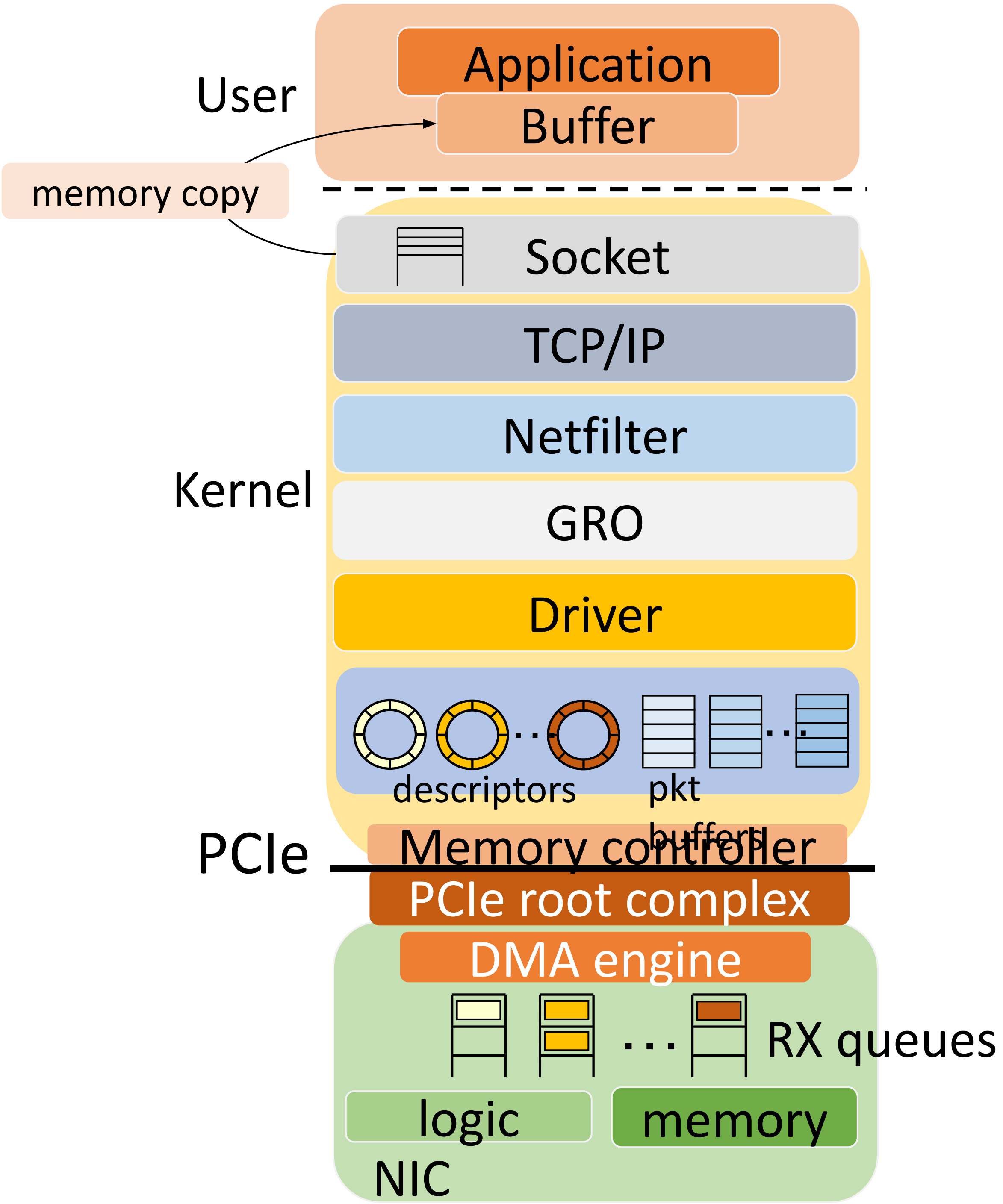
Kernel to User Space



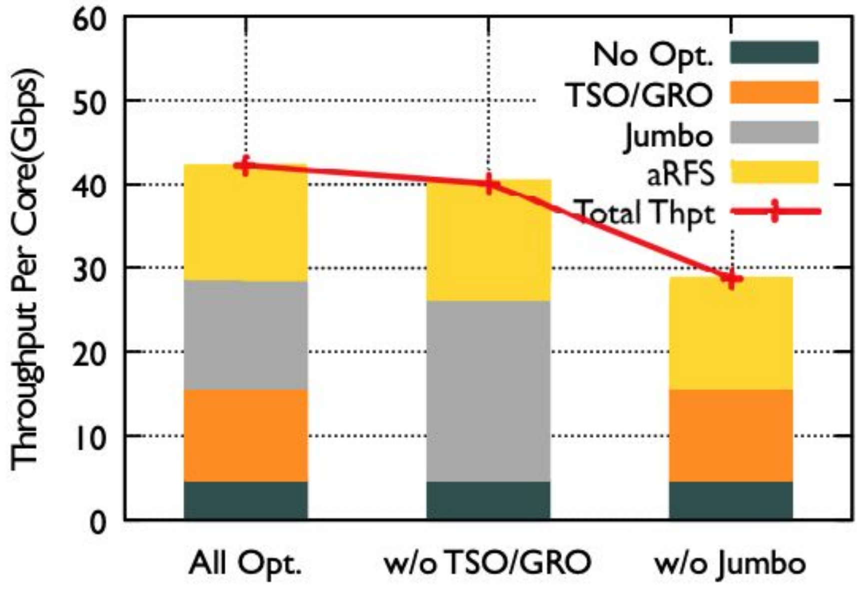
Important: data copy of packet payloads is performed only now. Operations within the kernel are done using metadata and pointer manipulations on skbs, and do not require data copy.

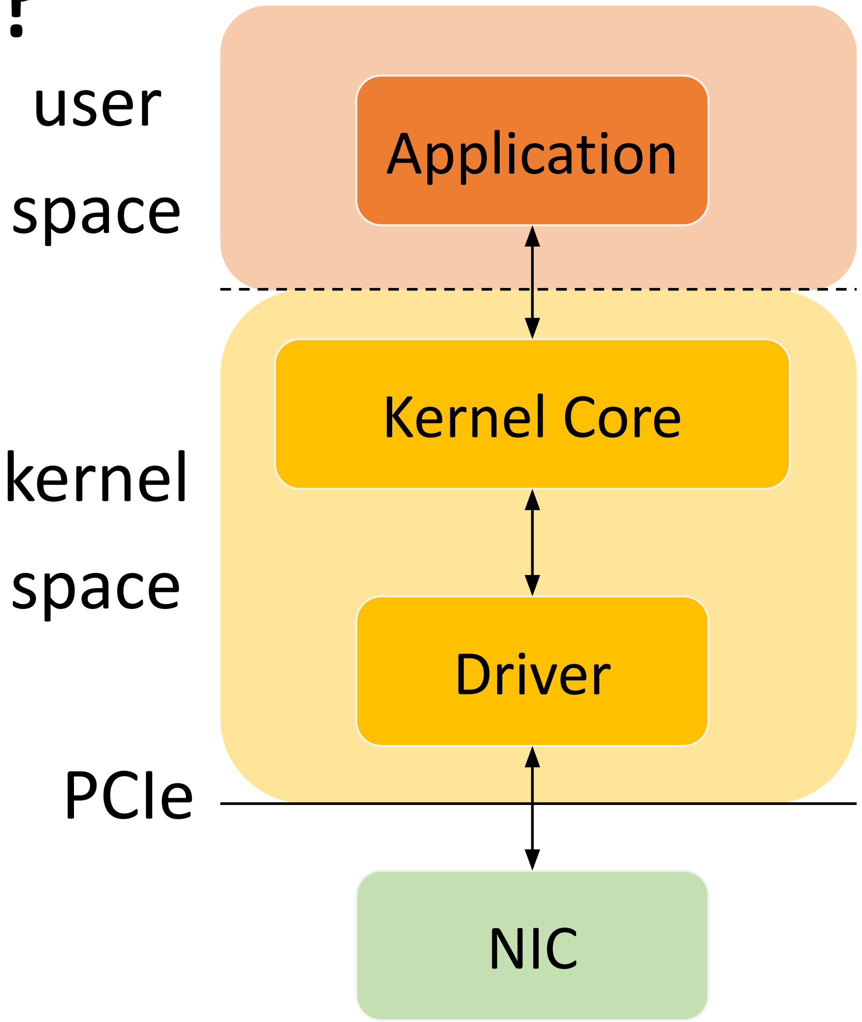
The kernel pkt buffer is given back to the descriptor pool.

putting this altogether



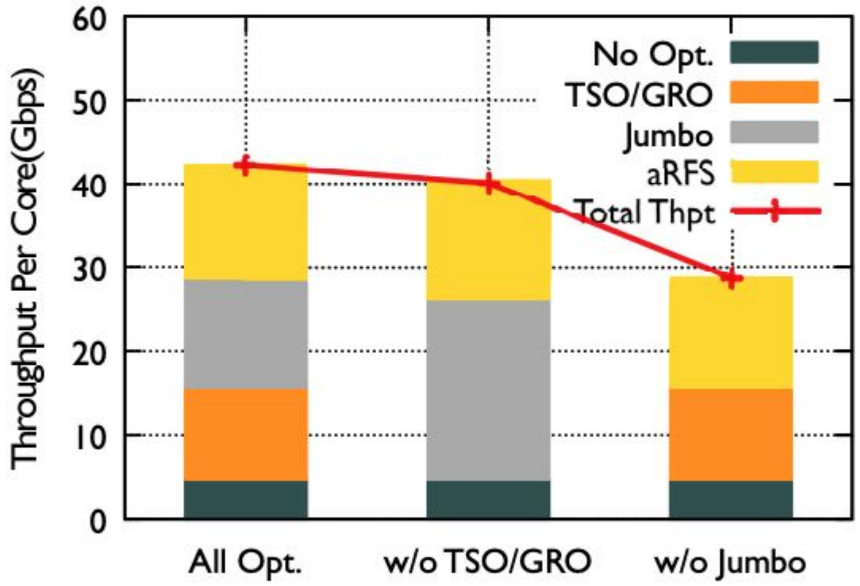
Is the network stack expensive?



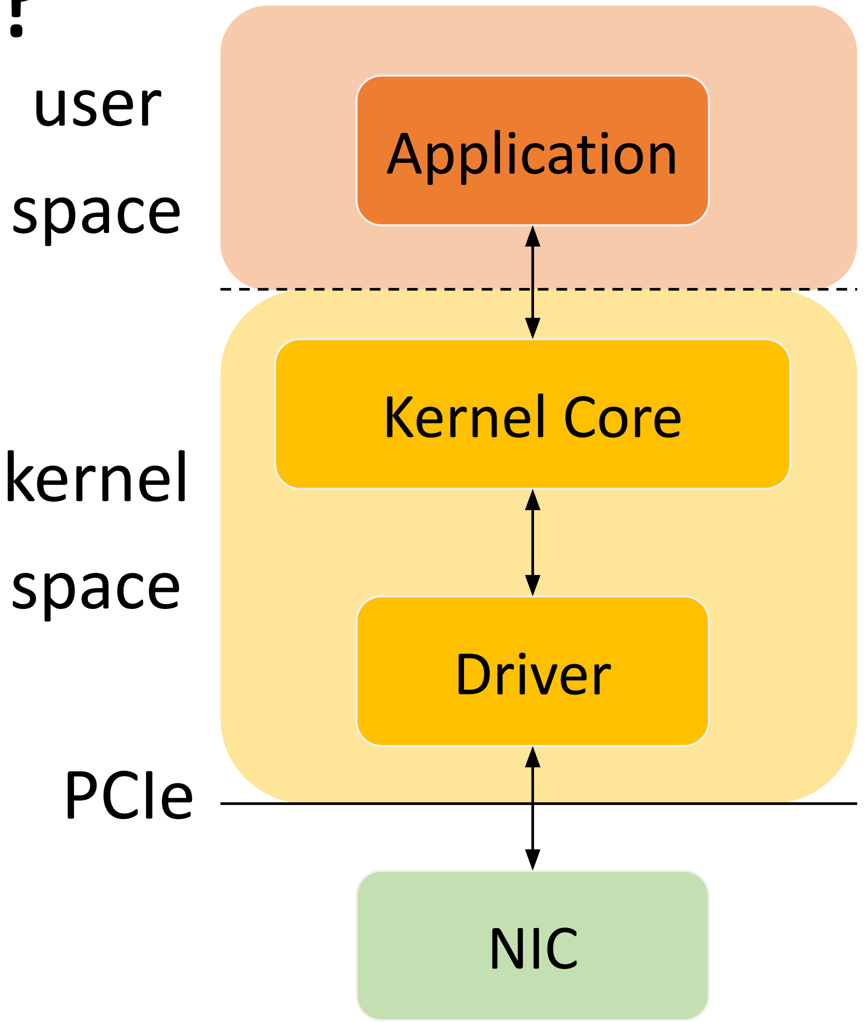


- A single core process up to 42Gbps
- TSO/GRO and Jumbo helps because they reduce the number of skbs

Is the network stack expensive?

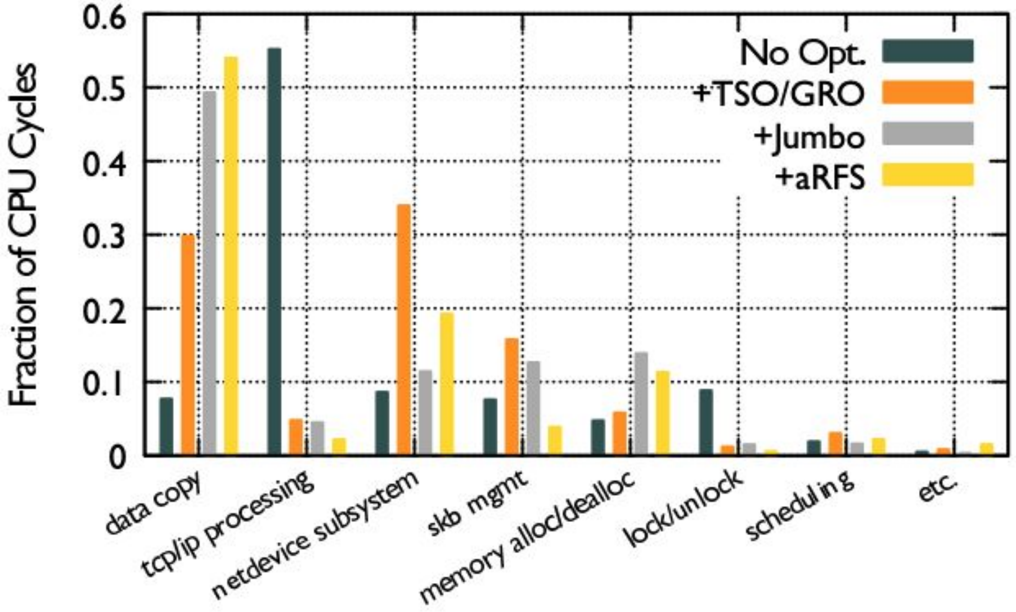


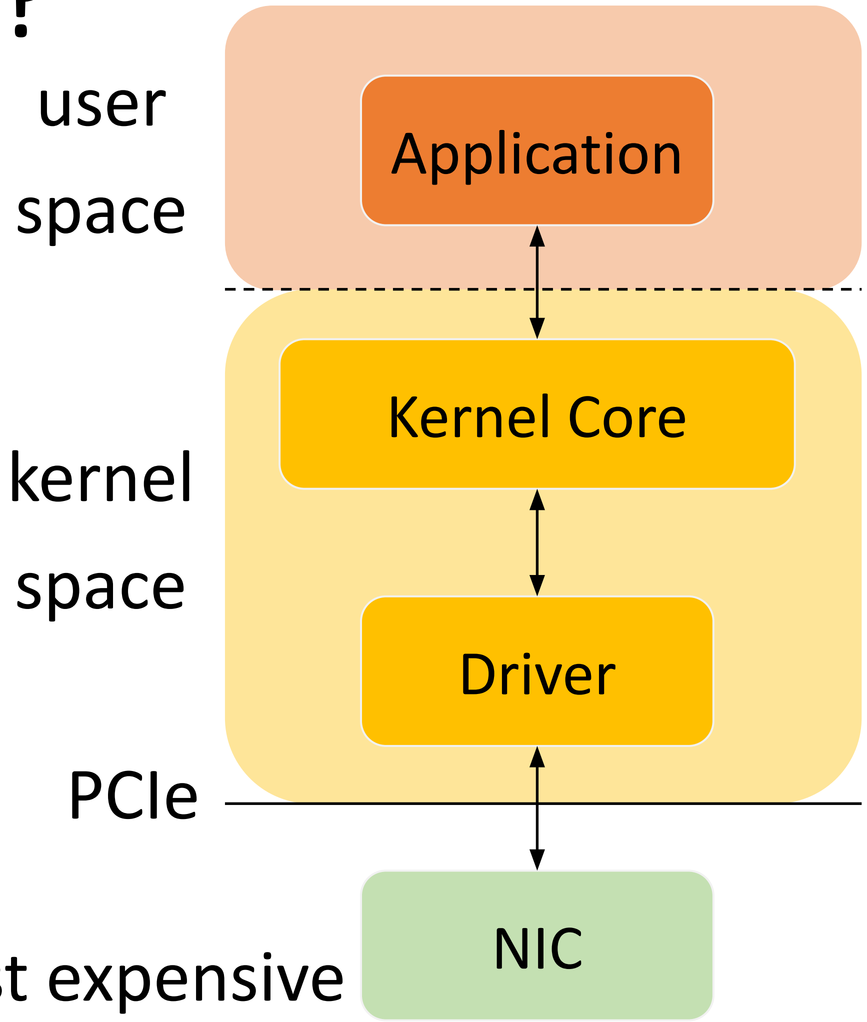
skbs allocation and processing is expensive!



- A single core process up to 42Gbps
- TSO/GRO and Jumbo helps because they reduce the number of skbs

Is the network stack expensive?





- No optimizations: TCP/IP processing is the most expensive
- All optimization: the data copy is the most expensive

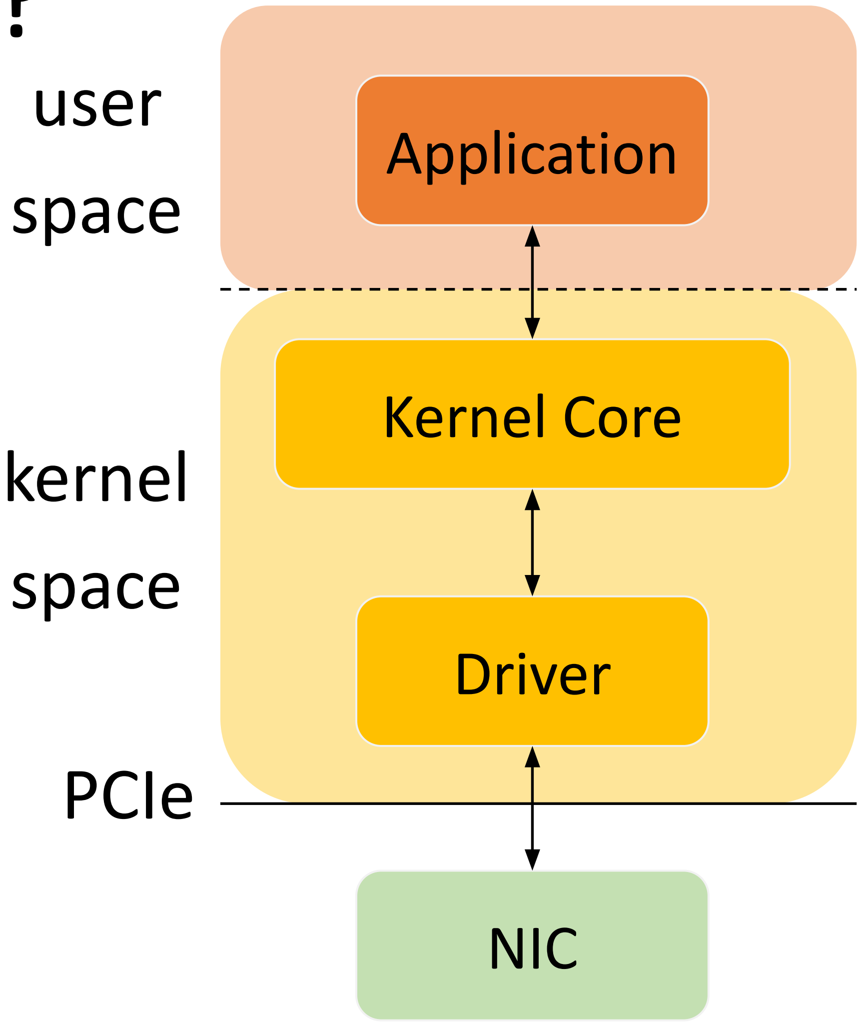
Is the network stack expensive?

It is also super complex!

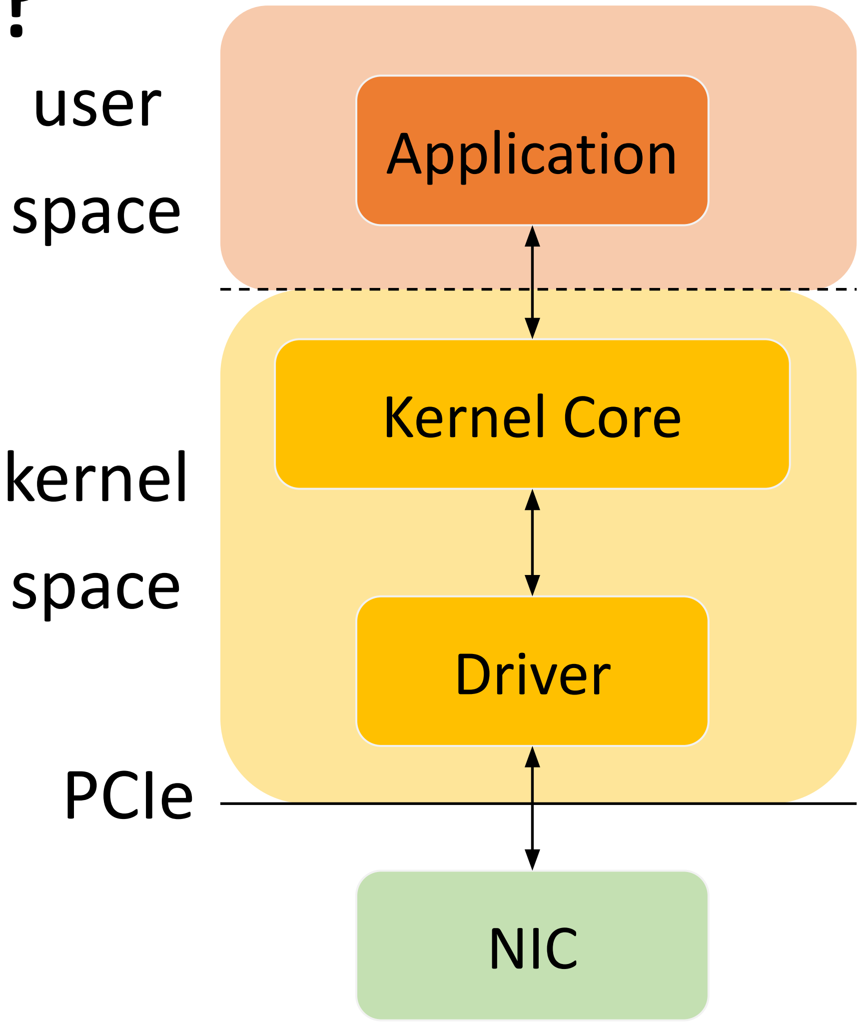
```
From: Coco Li <lixiaoyan@google.com>
To: Jakub Kicinski <kuba@kernel.org>,
        Eric Dumazet <edumazet@google.com>,
         Neal Cardwell <ncardwell@google.com>,
        Mubashir Adnan Qureshi <mubashirq@google.com>,
         Paolo Abeni <pabeni@redhat.com>, Andrew Lunn <andrew@lunn.ch>,
        Jonathan Corbet <corbet@lwn.net>,
         David Ahern <dsahern@kernel.org>,
        Daniel Borkmann <daniel@iogearbox.net>
Cc: netdev@vger.kernel.org, Chao Wu <wwchao@google.com>,
        Wei Wang <weiwan@google.com>,
         Pradeep Nemavat <pnemavat@google.com>,
        Coco Li <lixiaoyan@google.com>
Subject: [PATCH v8 net-next 0/5] Analyze and Reorganize core Networking Structs to optimize cacheline consumption
Date: Wed, 29 Nov 2023 07:27:51 +0000    [thread overview]
Message-ID: <20231129072756.3684495-1-lixiaoyan@google.com> (raw)

Currently, variable-heavy structs in the networking stack is organized
chronologically, logically and sometimes by cacheline access.

This patch series attempts to reorganize the core networking stack
variables to minimize cacheline consumption during the phase of data
transfer. Specifically, we looked at the TCP/IP stack and the fast
path definition in TCP.
```



Is the network stack expensive?



It is also super complex!

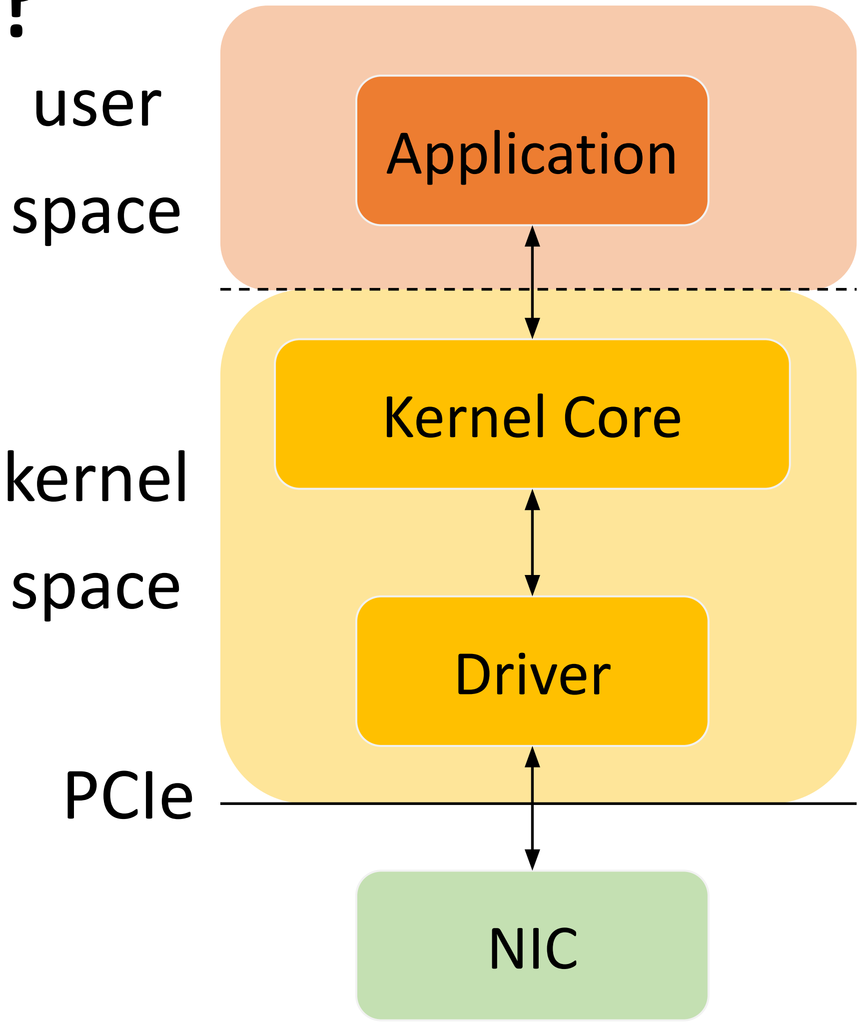
```
From: Coco Li <lixiaoyan@google.com>
To: Jakub Kicinski <kuba@kernel.org>,
        Eric Dumazet <edumazet@google.com>,
         Neal Cardwell <ncardwell@google.com>,
        Mubashir Adnan Qureshi <mubashirq@google.com>,
         Paolo Abeni <pabeni@redhat.com>, Andrew Lunn <andrew@lunn.ch>,
        Jonathan Corbet <corbet@lwn.net>,
         David Ahern <dsahern@kernel.org>,
        Daniel Borkmann <daniel@iogearbox.net>
Cc: netdev@vger.kernel.org, Chao Wu <wwchao@google.com>,
        Wei Wang <weiwan@google.com>,
         Pradeep Nemavat <pnemavat@google.com>,
        Coco Li <lixiaoyan@google.com>
Subject: [PATCH v8 net-next 0/5] Analyze and Reorganize core Networking Structs to optimize cacheline consumption
Date: Wed, 29 Nov 2023 07:27:51 +0000   [thread overview]
Message-ID: <20231129072756.3684495-1-lixiaoyan@google.com> (raw)

Currently, variable-heavy structs in the networking stack is organized
chronologically, logically and sometimes by cacheline access.

This patch series attempts to reorganize the core networking stack
variables to minimize cacheline consumption during the phase of data
transfer. Specifically, we looked at the TCP/IP stack and the fast
path definition in TCP.
```

```
On AMD platforms with 100Gb/s NIC and 256Mb L3 cache:
IPv4
Flows    with patches    clean kernel      Percent reduction
30k      0.0001736538065 0.0002741191042 -36.65%
20k      0.0001583661752 0.0002712559158 -41.62%
10k      0.0001639148817 0.0002951800751 -44.47%
5k       0.0001859683866 0.0003320642536 -44.00%
1k       0.0002035190546 0.0003152056382 -35.43%
```

Is the network stack expensive?



It is also super complex!

```
From: Coco Li <lixiaoyan@google.com>
To: Jakub Kicinski <kuba@kernel.org>,
        Eric Dumazet <edumazet@google.com>,
         Neal Cardwell <ncardwell@google.com>,
        Mubashir Adnan Qureshi <mubashirq@google.com>,
         Paolo Abeni <pabeni@redhat.com>, Andrew Lunn <andrew@lunn.ch>,
        Jonathan Corbet <corbet@lwn.net>,
         David Ahern <dsahern@kernel.org>,
        Daniel Borkmann <daniel@iogearbox.net>
Cc: netdev@vger.kernel.org, Chao Wu <wwchao@google.com>,
        Wei Wang <weiwan@google.com>,
         Pradeep Nemavat <pnemavat@google.com>,
        Coco Li <lixiaoyan@google.com>
Subject: [PATCH v8 net-next 0/5] Analyze and Reorganize core Networking Structs to optimize cacheline consumption
Date: Wed, 29 Nov 2023 07:27:51 +0000    [thread overview]
Message-ID: <20231129072756.3684495-1-lixiaoyan@google.com> (raw)

Currently, variable-heavy structs in the networking stack is organized
chronologically, logically and sometimes by cacheline access.

This patch series attempts to reorganize the core networking stack
variables to minimize cacheline consumption during the phase of data
transfer. Specifically, we looked at the TCP/IP stack and the fast
path definition in TCP.
```

On AMD platforms with 100Gb/s NIC and 256Mb L3 cache:
IPv4

Flows	with patches	clean kernel	Percent reduction
30k	0.0001736538065	0.0002741191042	-36.65%
20k	0.0001583661752	0.0002712559158	-41.62%
10k	0.0001639148817	0.0002951800751	-44.47%
5k	0.0001859683866	0.0003320642536	-44.00%
1k	0.0002035190546	0.0003152056382	-35.43%

can we do better? ☺️