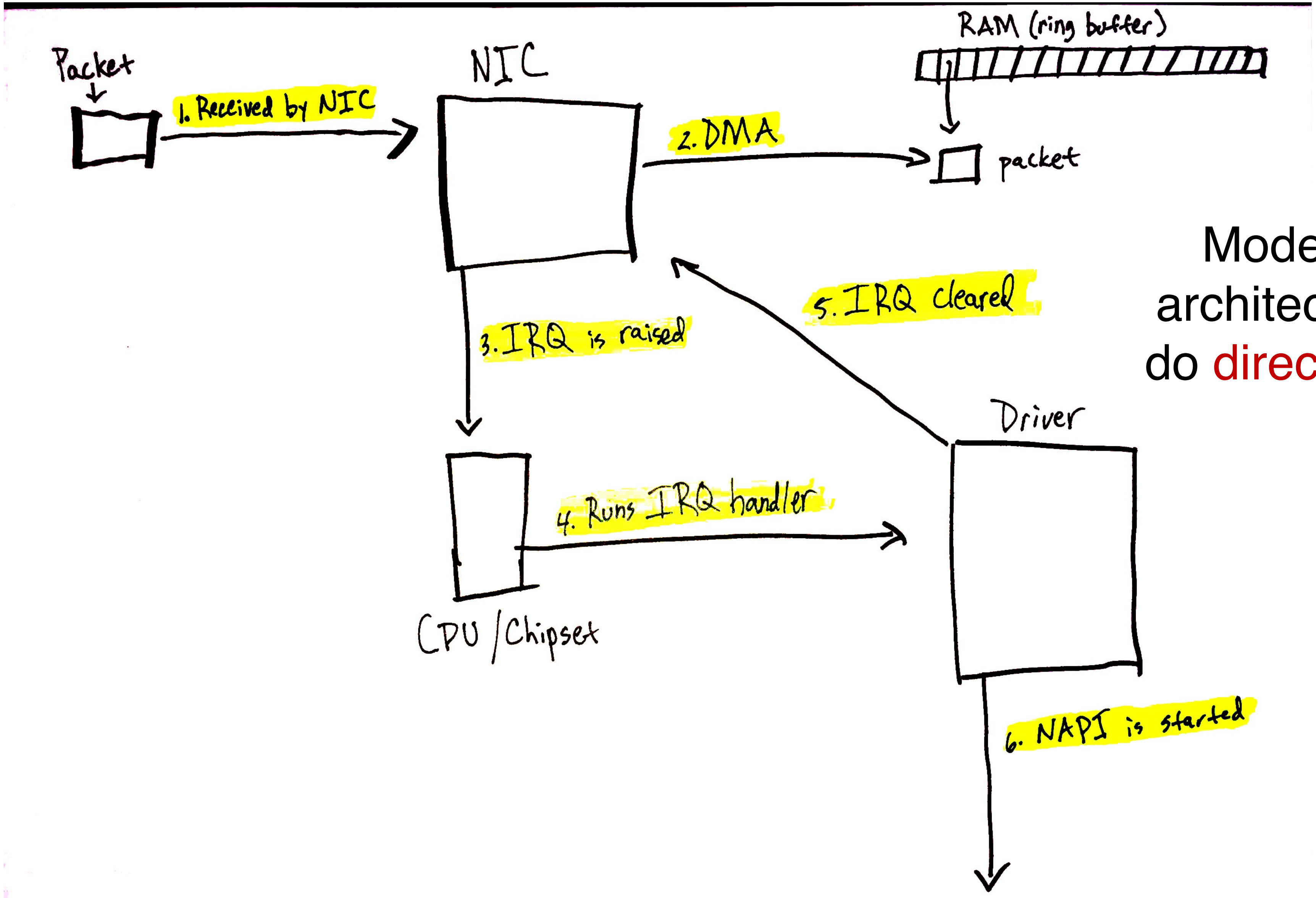
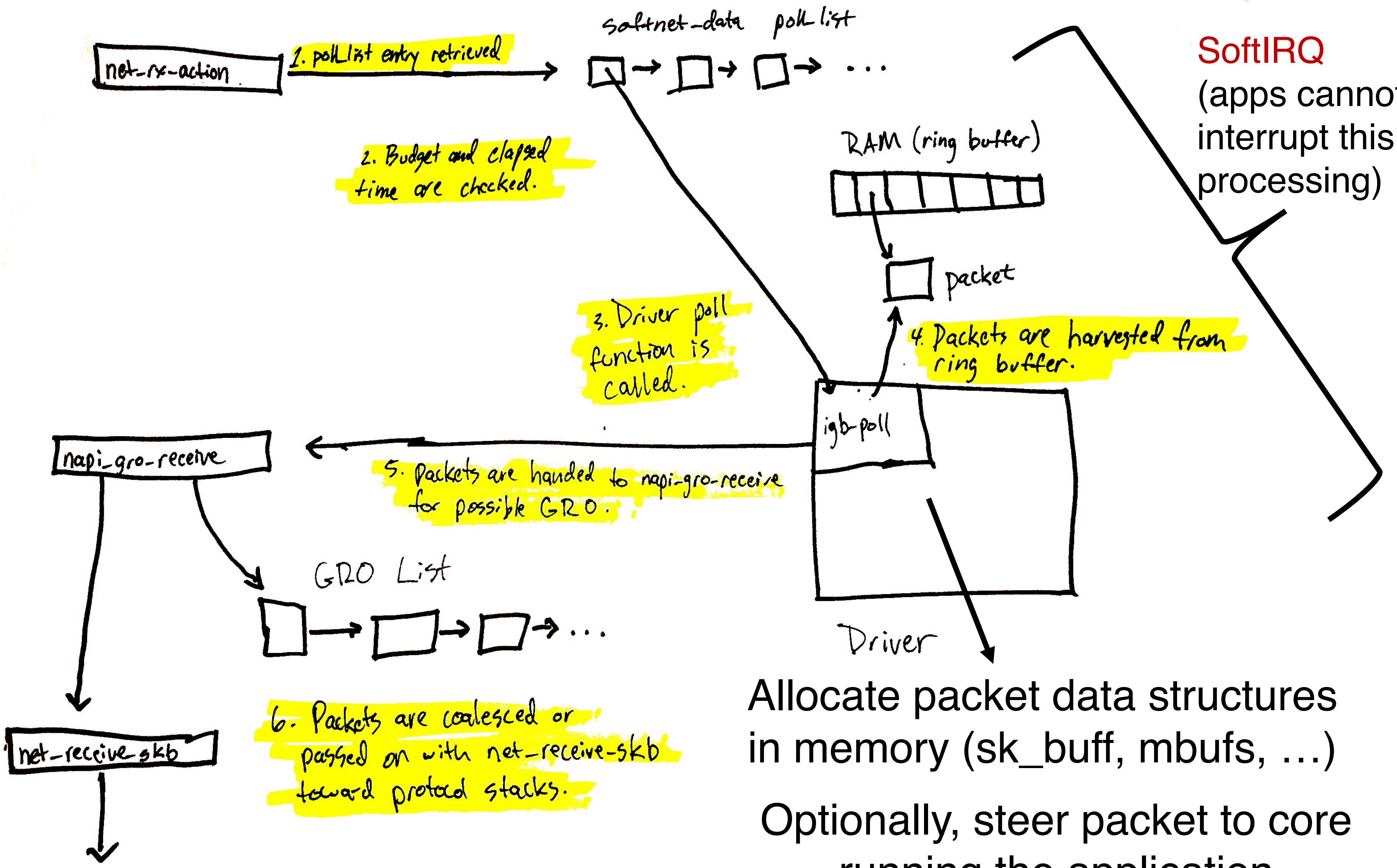
Network



Modern NICs and architectures can also do direct cache access (DCA)

Interrupt mitigation

- Interrupt processing at high rate and priority prevents any other part of the system from progressing (receive livelock).
- Mitigations:
- (1) Interrupt coalescing:
  - Wait (at NIC) for more packets or a timeout until interrupting
- (2) Polling to schedule the work, avoiding preemption
- (3) CPU or packet quotas on polling to ensure other parts of the system (e.g. user space app) can progress
  - Re-enable interrupts if there is less work than allotted quota



SoftIRQ
(apps cannot interrupt this processing)

Allocate packet data structures in memory (sk_buff, mbufs, …)

Optionally, steer packet to core running the application

Other things that happen afterward

- Netfilter: tracking TCP connection state, firewalling, NAT, tcpdump
- IP protocol processing: routing
- Transport processing (UDP/TCP protocol layer)
- Copy into user space socket buffers
- Applications use socket APIs to process the packets

- Work that is independent per (group of) packets can often be handed off to the NIC. These are often referred to as NIC offload
  - TSO: TCP segmentation offload; LRO: Large Receive Offload
  - IP checksum (transmit & receive)
  - https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html

Different Kinds of Packet Steering

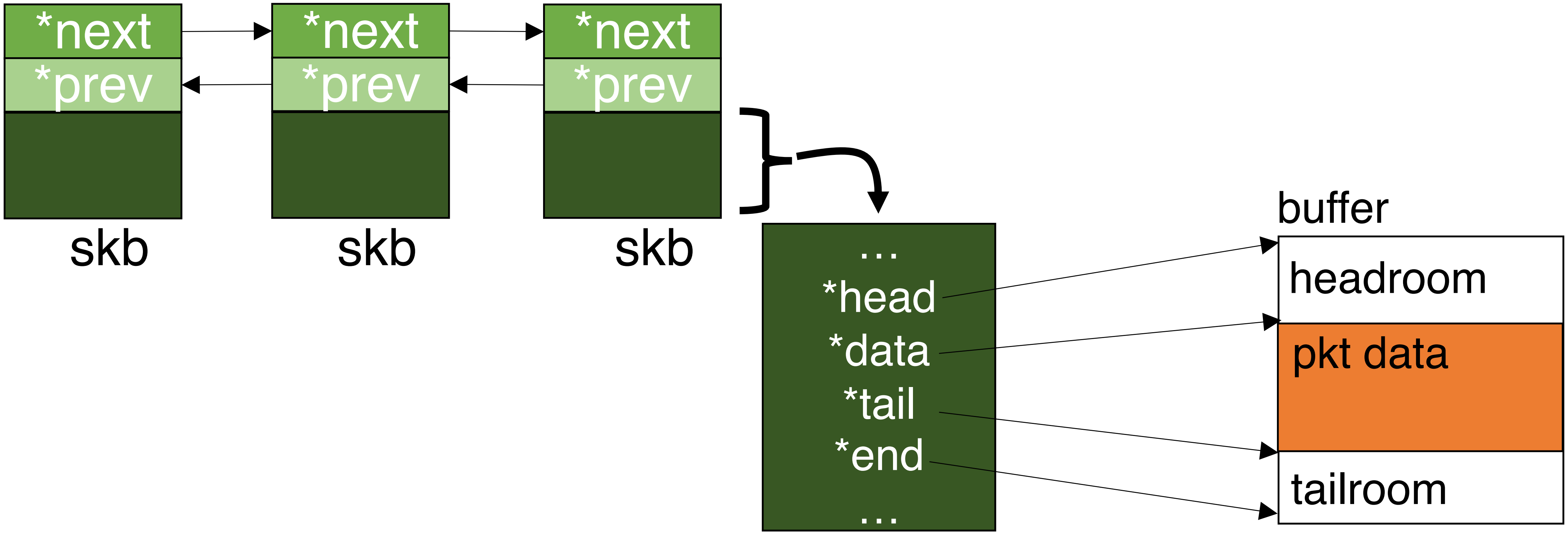
- Receive-Side Scaling (RSS)
  - NIC determines which CPU to hardware interrupt (IRQ)
  - NICs have multiple queues, process each queue (potentially) at different CPU cores
  - Use a hash function over packet headers at NIC to direct to cores
  - Each NIC receive queue has a different associated IRQ #
  - IRQs can be configured to have affinity to specific CPU cores
  - This CPU runs the hardware interrupt handler
- Receive Packet Steering (RPS)
  - select CPU to handle protocol processing after interrupt handling (starting from netif_receive_skb). Use inter-processor interrupts
  - Useful as a pure software method to distribute protocol processing

Different Kinds of Packet Steering

- Receive Flow Steering (RFS)
  - Like RPS, but consider the CPU where the application is running
  - Improve data cache hit rates (packet read on the same CPU core that it was written to)
  - Pure software technique
- Accelerated Receive Flow Steering (aRFS)
  - Implement RFS affinity in hardware
  - Network stack identifies which CPU is processing packets
  - Device driver programs the appropriate queue # into hardware
  - Needs hardware support

Socket buffers

- Allocate packet data in arbitrary chunks (multiples of 64 bytes)
- Support arbitrary packet sizes, fragments, deferred processing



FreeBSD sendto() code path

Overheads are sprinkled throughout the packet processing stack.

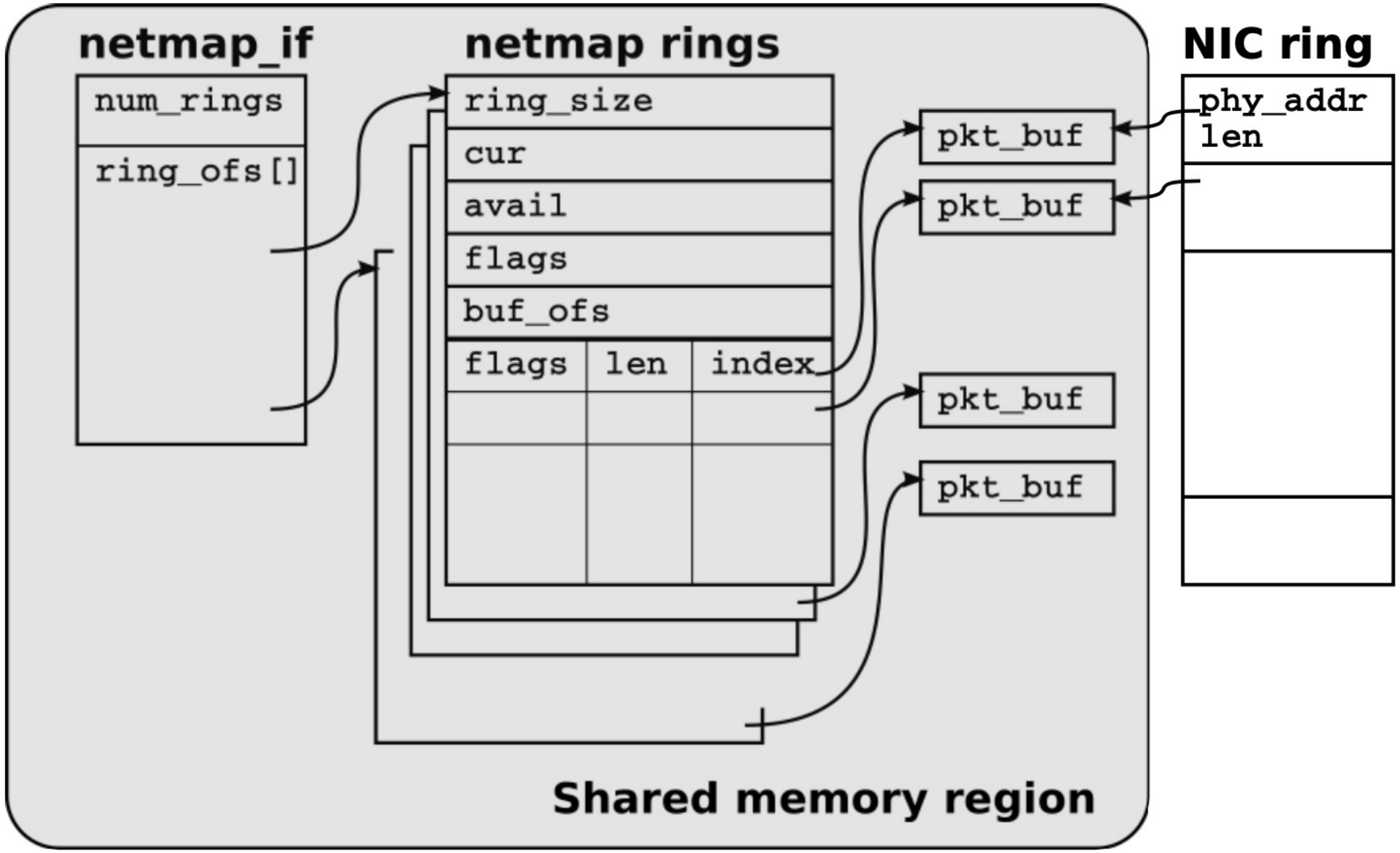
Software specialization

File	Function/description	time ns	delta ns
user program	sendto system call	8	96
uipc_syscalls.c	sys_sendto	104	
uipc_syscalls.c	sendit	111	
uipc_syscalls.c	kern_sendit	118	
uipc_socket.c	sosend	—	
uipc_socket.c	sosend_dgram sockbuf locking, mbuf allocation, copyin	146	137
udp_usrreq.c	udp_send	273	
udp_usrreq.c	udp_output	273	57
ip_output.c	ip_output route lookup, ip header setup	330	198
if_ethersubr.c	ether_output MAC header lookup and copy, loopback	528	162
if_ethersubr.c	ether_output_frame	690	
ixgbe.c	ixgbe_mq_start	698	
ixgbe.c	ixgbe_mq_start_locked	720	
ixgbe.c	ixgbe_xmit mbuf mangling, device programming	730	220
–	on wire	950	

Netmap ATC12.

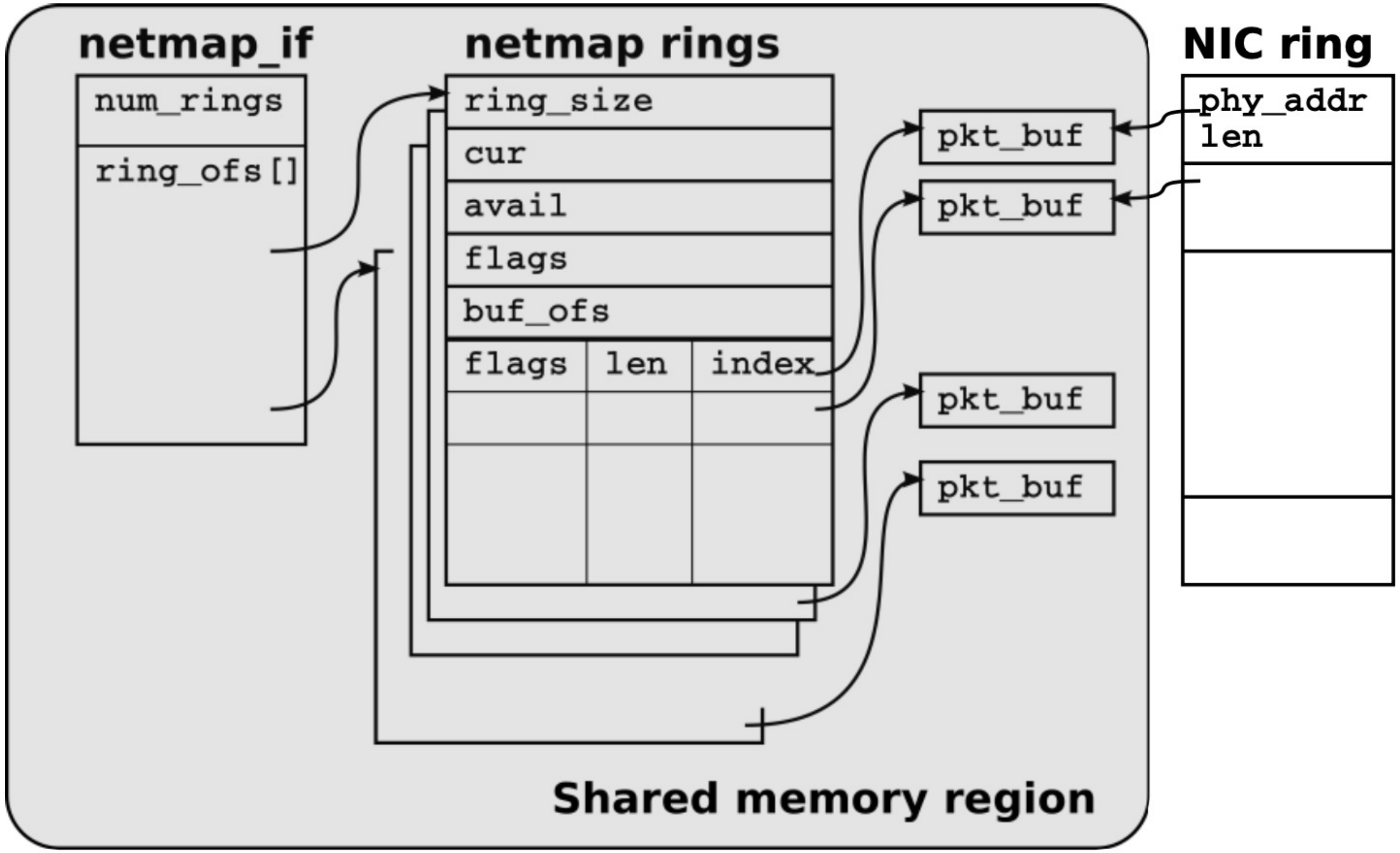
(1) Shared memory: avoid per-byte costs

- Remove user-kernel data copies
- Other systems use similar ideas:
- Finish processing entirely within the kernel (e.g., click-kernel, eBPF)
  - Expressiveness
- Expose NIC buffers directly to user space (PF_RING, DPDK)
  - Isolation



(2) Data representation: pre-allocated fixed size buffers and rings

- Avoid per-byte costs by pre-allocating chunks of a fixed size (max packet size)
- No allocation and freeing mbuf/sk_buff at run time
- Exchange descriptors on ring buffers between app and NIC



(3) Amortize operations: batching

- Batch notifications to NIC for packets written for transmission or free buffers available for reception.
- Process packets in batches
- Effect: Better instruction cache hit rates

```
for (;;) {
        /*
         * Receive packets on a port and forward them on the paired
         * port. The mapping is 0 -> 1, 1 -> 0, 2 -> 3, 3 -> 2, etc.
         */
        RTE_ETH_FOREACH_DEV(port) {

                /* Get burst of RX packets, from first port of pair. */
                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;

                /* Send burst of TX packets, to second port of pair. */
                const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                                bufs, nb_rx);

                /* Free any unsent packets. */
                if (unlikely(nb_tx < nb_rx)) {
                        uint16_t buf;
                        for (buf = nb_tx; buf < nb_rx; buf++)
                                rte_pktmbuf_free(bufs[buf]);
                }
        }
}
```

The abstraction has changed!

- The techniques above (embodied into fast packet processing frameworks like netmap, DPDK, eBPF) aim to move data to applications quickly
    - Ideal for middleboxes and software routers

- But if needed, applications must re-implement functionality that is already part of the kernel network stack (e.g. transport)
    - The benefit of these frameworks is less clear for application endpoints which *do* need transport, routing, …

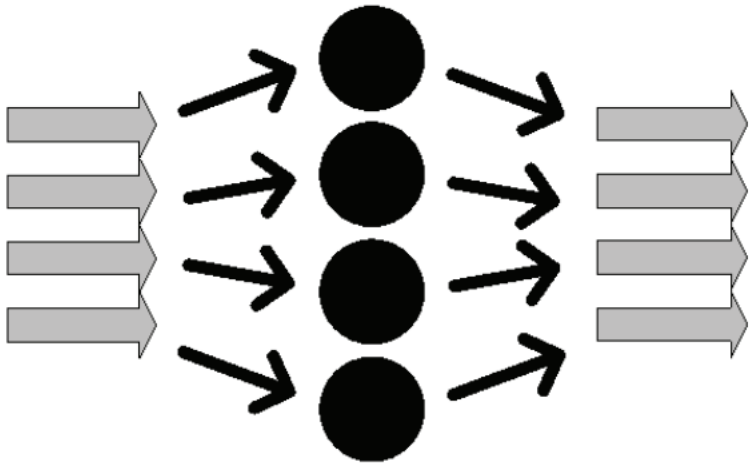
- Typical utilities (ping, tcpdump, etc.) may no longer work

- The story becomes more complicated with virtualization

Case studies

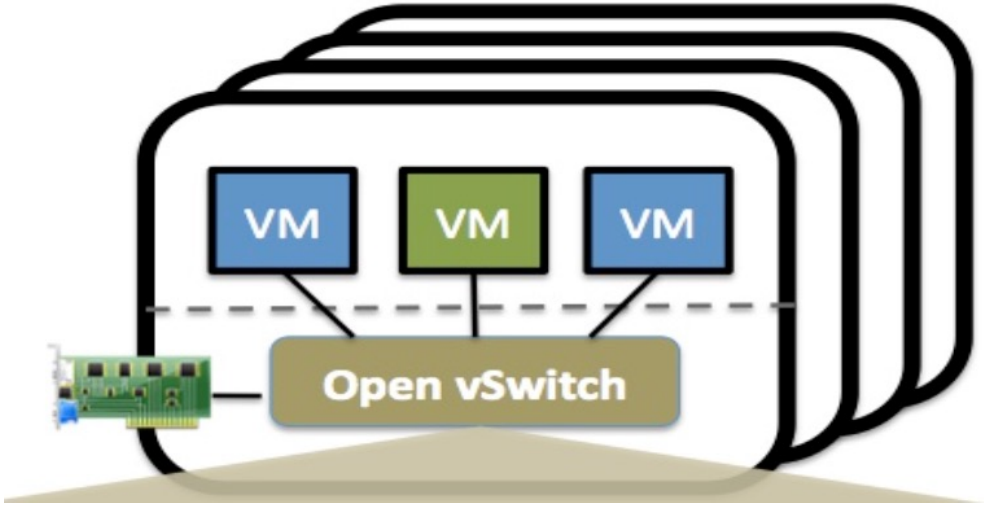
Routebricks: fast software router

- Inspiration from interconnects
- Fast processing on a single machine
- Multi-queue NICs
- Data interconnection patterns between queues and cores
    - Receive side scaling (RSS)



OpenVSwitch: fast virtual switch

- Early roots in networking: first switches were fully in software
  - Until high link speeds forced everyone to make ASICs
- As a tool for experimentation with SDN protocols (eg: Openflow)
- Advent of virtualization
  - Need flexible policies (ie: flow rules) inside endpoints!

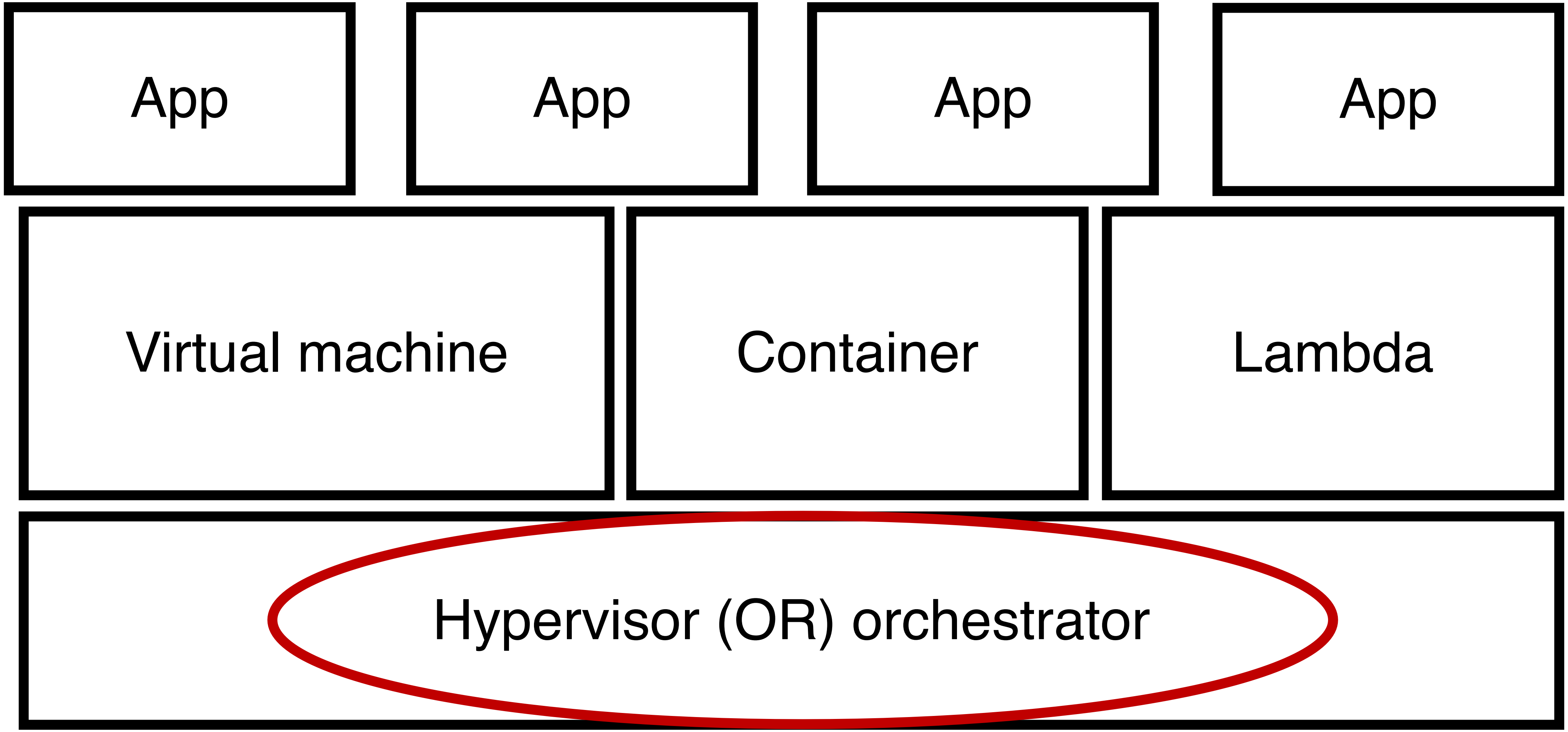


https://www.openvswitch.org/

Policies in virtualized switches

- Tenant policies
  - Network virtualization: I want the physical network to look like my own, and nobody else is on it. Use own addresses
- Provider policies
  - Traffic must follow the ACLs and paths set by the provider
- Topology traversal
  - Use the core of the DCN as a mesh of point to point tunnels

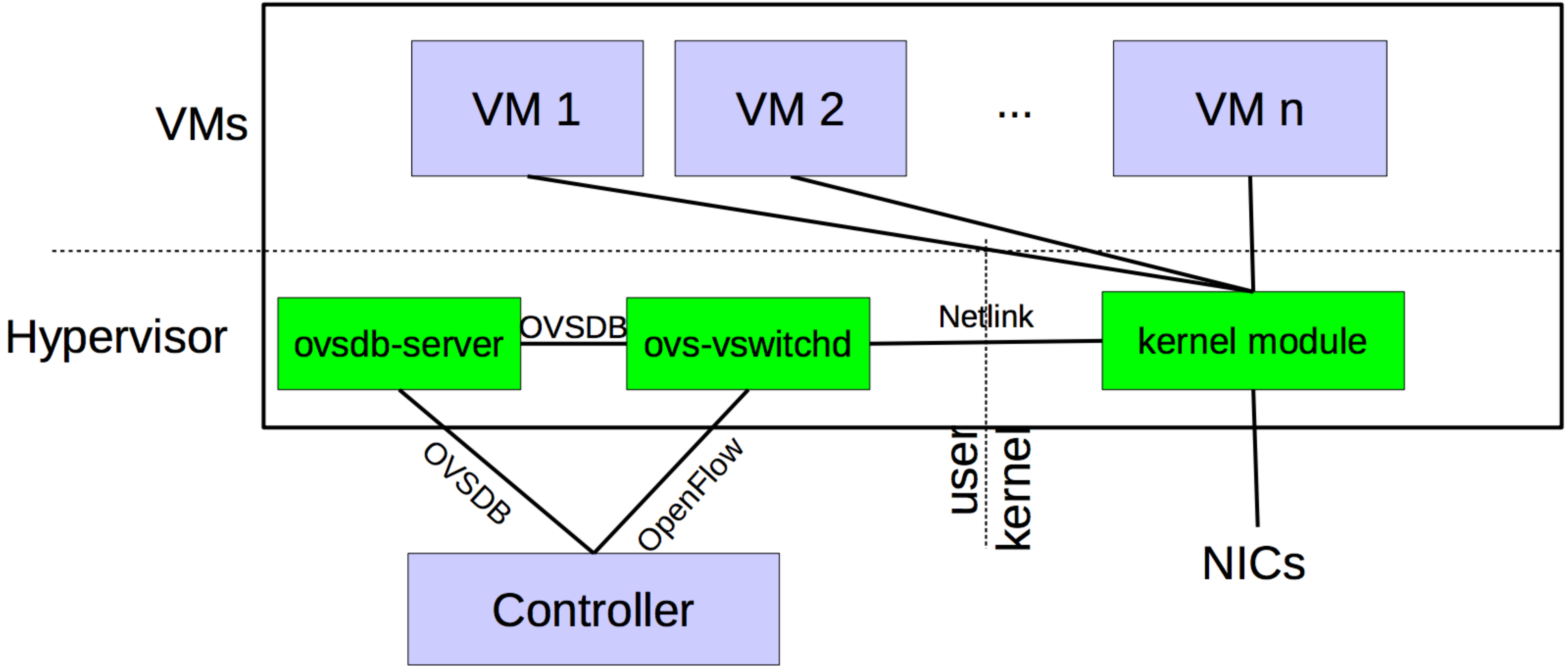
Where should policies be implemented?



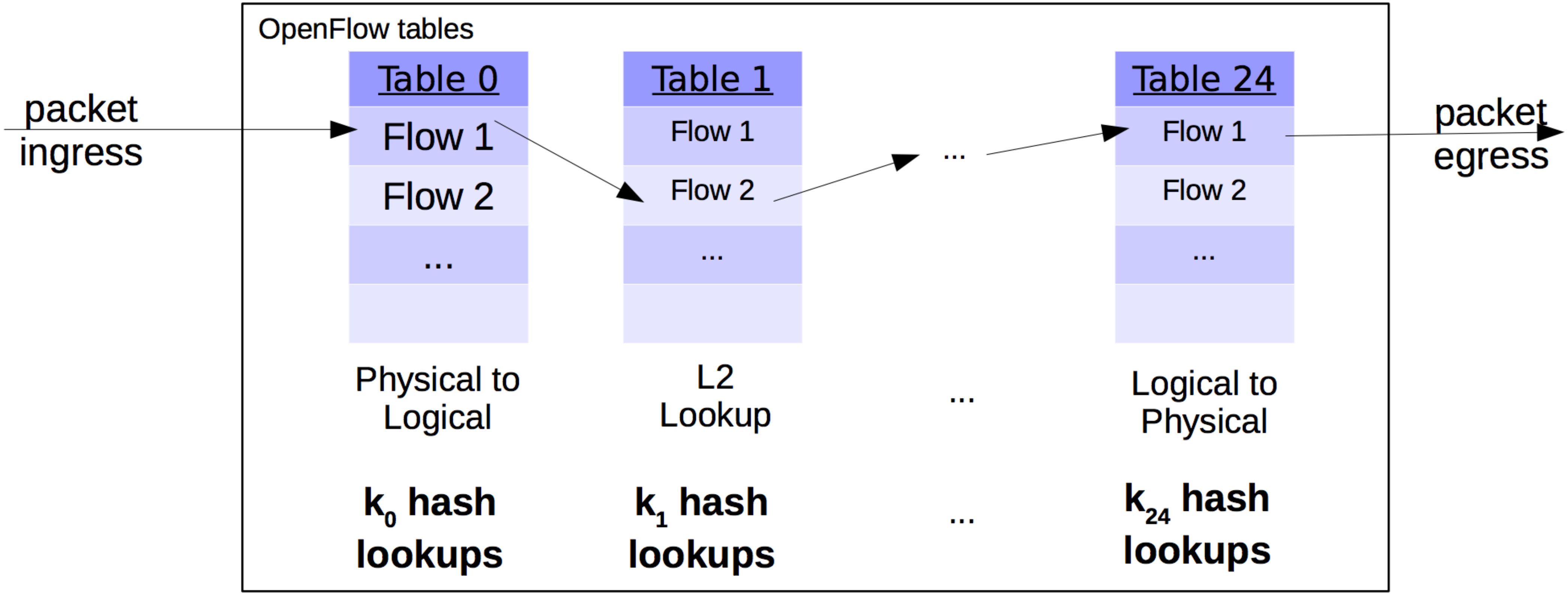
OpenVSwitch: Requirements

- Support large and complex policies
- Support updates in such policies, e.g., VM migration, new customers, …
- Don’t take up too much resources (CPU must do useful work, not just policy processing)
- Process packets with high performance
  - High throughput and low delay

OVS design


VMs
VM 1
VM 2
...
VM n
Hypervisor
ovsdb-server
OVSDB
ovs-vswitchd
Netlink
kernel module
OVSDB
OpenFlow
user
kernel
NICs
Controller

First design: put OF tables in the kernel



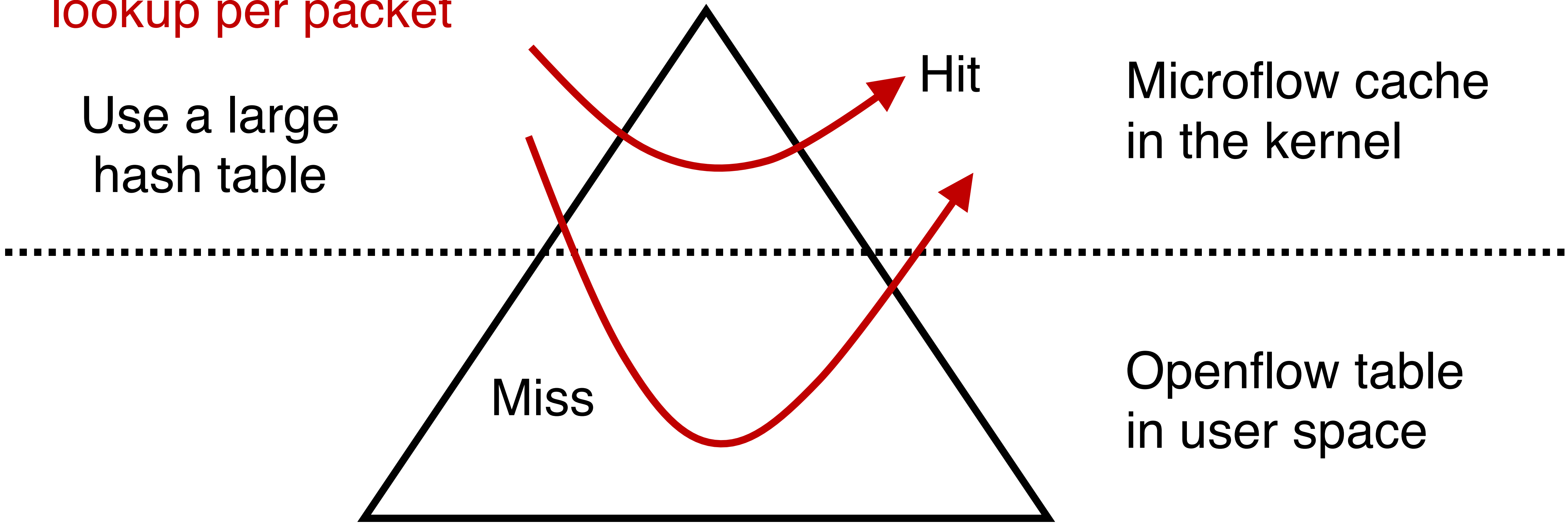
Large policies: Low performance with 100+ lookups per packet

Merging policies is problematic: cross-product explosion

Complex logic in kernel: rules with wildcards require complex algorithms

Idea 1: Microflow cache

- Microflow: complete set of packet headers with action
  - Example: srcIP, dstIP, IP TTL, srcMAC, dstMAC
- Same insight as tuple space search; attempt to do one memory lookup per packet



Problems with micro-flows

- Too many micro-flows: e.g., each TCP port
- Many micro-flows may be short lived
  - Poor cache-hit rate for memory lookup

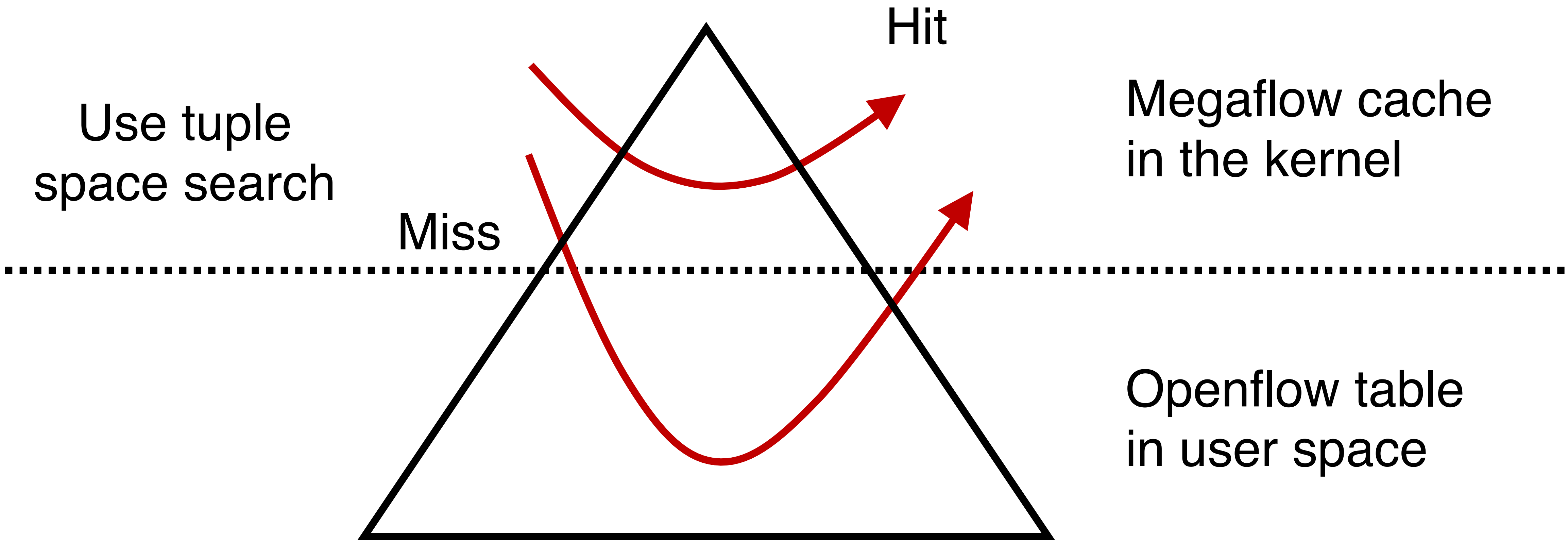
- Can we cache the outcome of rule lookup directly?

- Naive approach: Cross-product explosion!
  - Example: Table 1 on source IP, table 2 on destination IP

- Recurring theme: avoid up-front (proactive) costs

Idea 2: Mega-flow cache

- Build the cache of rules lazily using just the fields accessed
  - Ex: contain just src/dst IP combinations that appeared in packets



Outlook: fast packet processing

- Get rid of needless software if you can
- Specialization to app can bring significant benefits
  - IDS (hyperscan), caching in switches & load balancers
  - Algorithms can be as important as the frameworks
- Software changes
  - Application-kernel interface: application must be modified
  - Device drivers must often be modified
- Multitenancy: think about implications to weakening fault isolation
- Can we get isolation with efficiency?

Going beyond one (software) box

- Safe & efficient composition of middleboxes

- Share or shard state
- Failover and migration
- Placement and routing
- Scaling and compaction