Programmable NICs

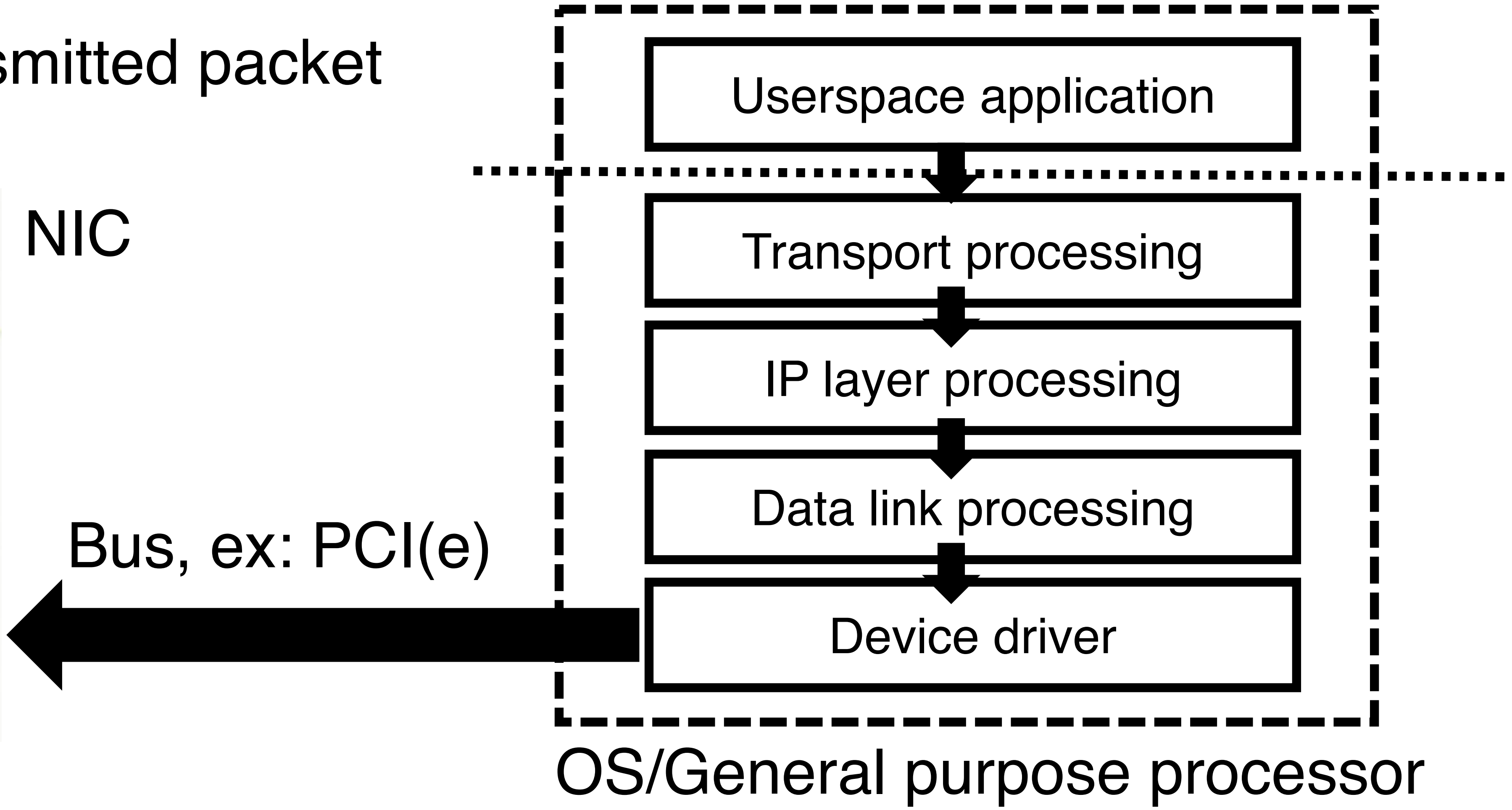
Lecture 14, Computer Networks (198:552)

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Network Interface Cards (NICs)

- The physical interface between a machine and the wire
- Life of a transmitted packet

NIC



Data transfers between the host & NIC

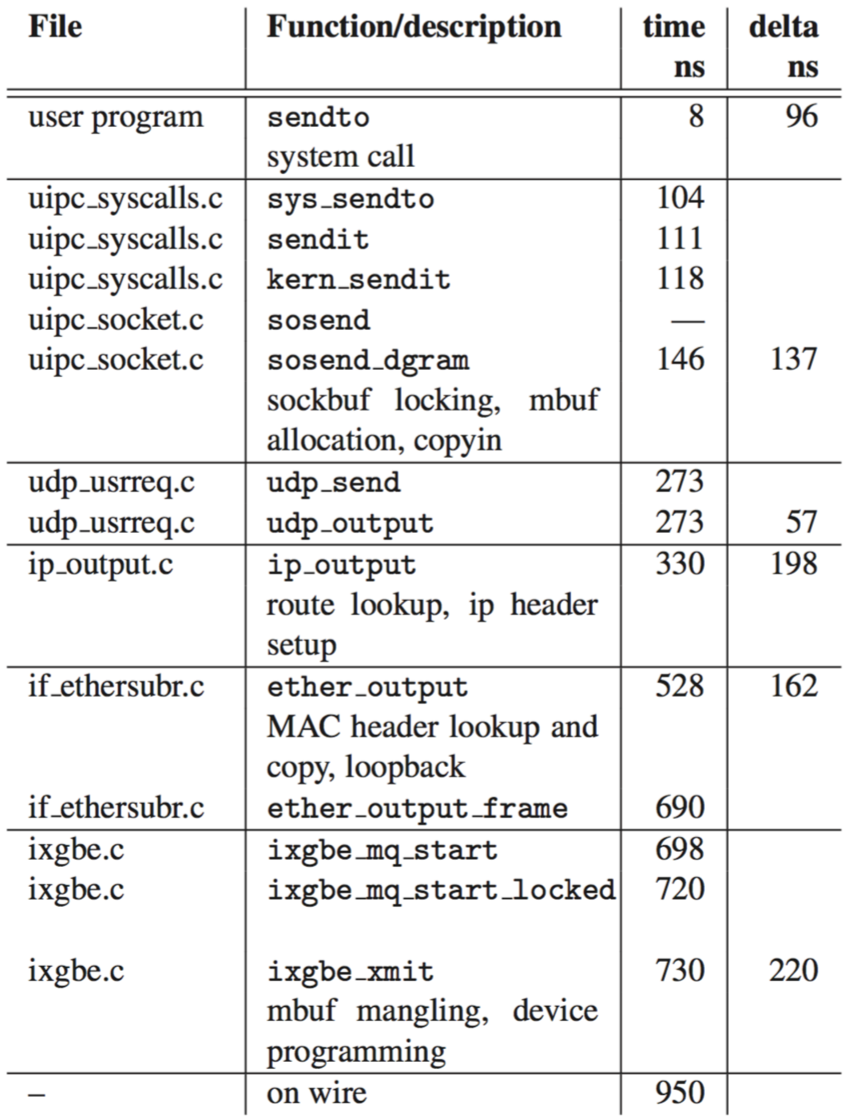
- Receive (*rx*) path
  - Polling by the CPU
  - Interrupt the CPU whenever a packet arrives
- Transmit (*tx*) path
  - Programmed input/output: CPU explicitly moves data
  - Direct Memory Access (DMA)
- Multiple queues to transmit and receive data
  - Allow scheduling/prioritization, isolation, and load balancing

NIC speeds have been going up…

- 2009: 1 Gbit/s
- 2012: 10 Gbit/s
- 2015: 40 Gbit/s
- 2018: 100 Gbit/s
- Soon: 200 and 400 Gbit/s standards

But OS stack has significant overheads!

- Measurements from [netmap'12]
  - FreeBSD, i7 (2.93 GHz), ixgbe driver
- Sources of overhead:
  - System call
  - Memory allocation
  - IP routing/header lookup
  - (surprisingly) MAC lookups
  - Moving data from
  device memory to NIC memory
- Affects throughput, delay, CPU util



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	

CPU processing won’t get much better

- The end of “Dennard scaling” and “Moore’s law”

- Transistor power dissipation isn’t going down with generations
  - Can’t clock x86 cores with higher frequencies
  - Implication: performance for serialized computation has stagnated

- Transistor feature sizes are approaching quantum limits
  - Number of transistors may increase in the near future
  - But it may stagnate pretty soon
  - Implication: there is only so much parallelism to be extracted

Approaches to keep up with high speed

1) Optimize the common case (in the existing network stack)
2) Harden common stateless operations in the NIC ASIC
3) Use parallelism by processing packets with multiple cores
4) Bypass the OS kernel entirely

----

5) Bypass the hypervisor entirely (in virtualized environments)
6) Offload packet processing to some high-perf substrate
    - ASIC, SoC, FPGAs

(1) Optimize the common case

- Fast and slow path separation
  - Ex: TCP ACK processing (e.g., 30 instructions by Van Jacobson)
  - Ex: IP route caching

- Batching:
  - NIC: Interrupt coalescing
  - NIC: Receive segment coalescing
  - Stack: Transmit packets in a bunch until no more to transmit

(2) Harden stateless operations in NIC

- Move some of the fixed per-packet *independent* processing to NIC
    - Historically the main reason why software kept up with link speeds

- Examples:
    - TCP and IP checksum computations
    - “Scatter gather” I/O
    - Segmentation offload (tx and rx)
    - … and many more

- Often built first in software and moved to NIC hardware

(3) Process packets with multiple cores

- Receive-Side Scaling (RSS): on packet receive,
  - Hash the packet into one of N buckets
  - Each bucket mapped to a core on the host
  - Interrupt the corresponding core
- May use separate queue for each core
- May separate the packets by connection
  - Single transport connection is the granularity of maintaining state

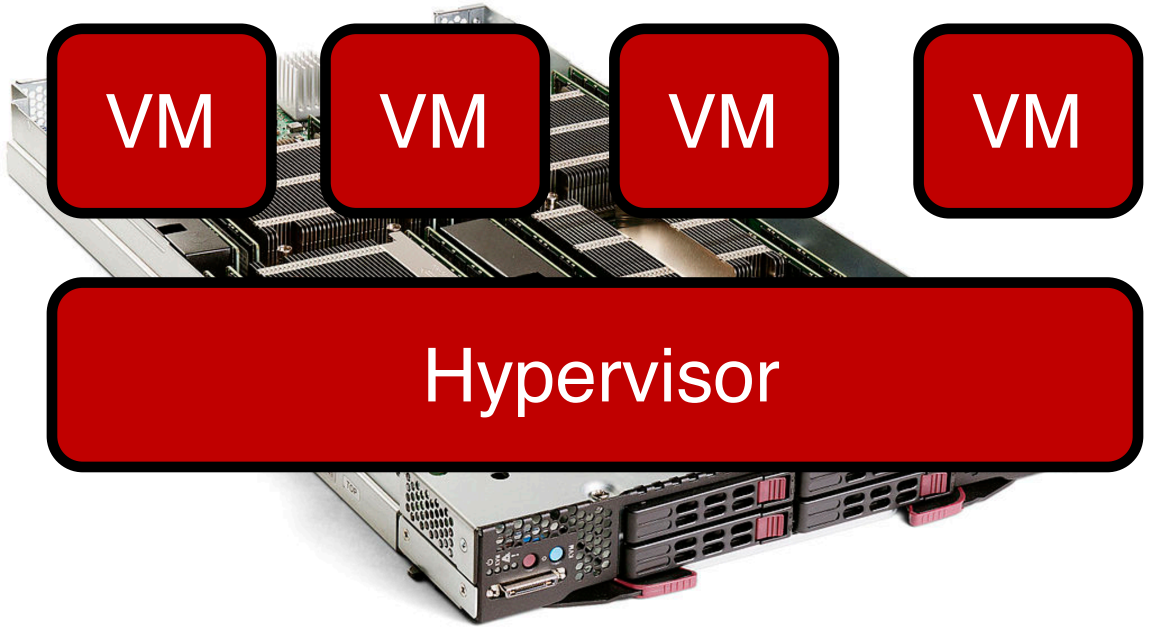
- Analogous: Transmit-side steering

(4) Bypass the OS kernel altogether

- Use a custom device driver that polls the network device
    - Give up a core or two simply to poll the NIC!
- Share memory directly between driver and user-space
    - Flexible way to implement many different features in user-space
- Intel’s Data Plane Development Kit (DPDK)
    - Many high performance “network functions”
- Cons: breaks OS abstractions and isolation between processes

Virtualized cluster environments

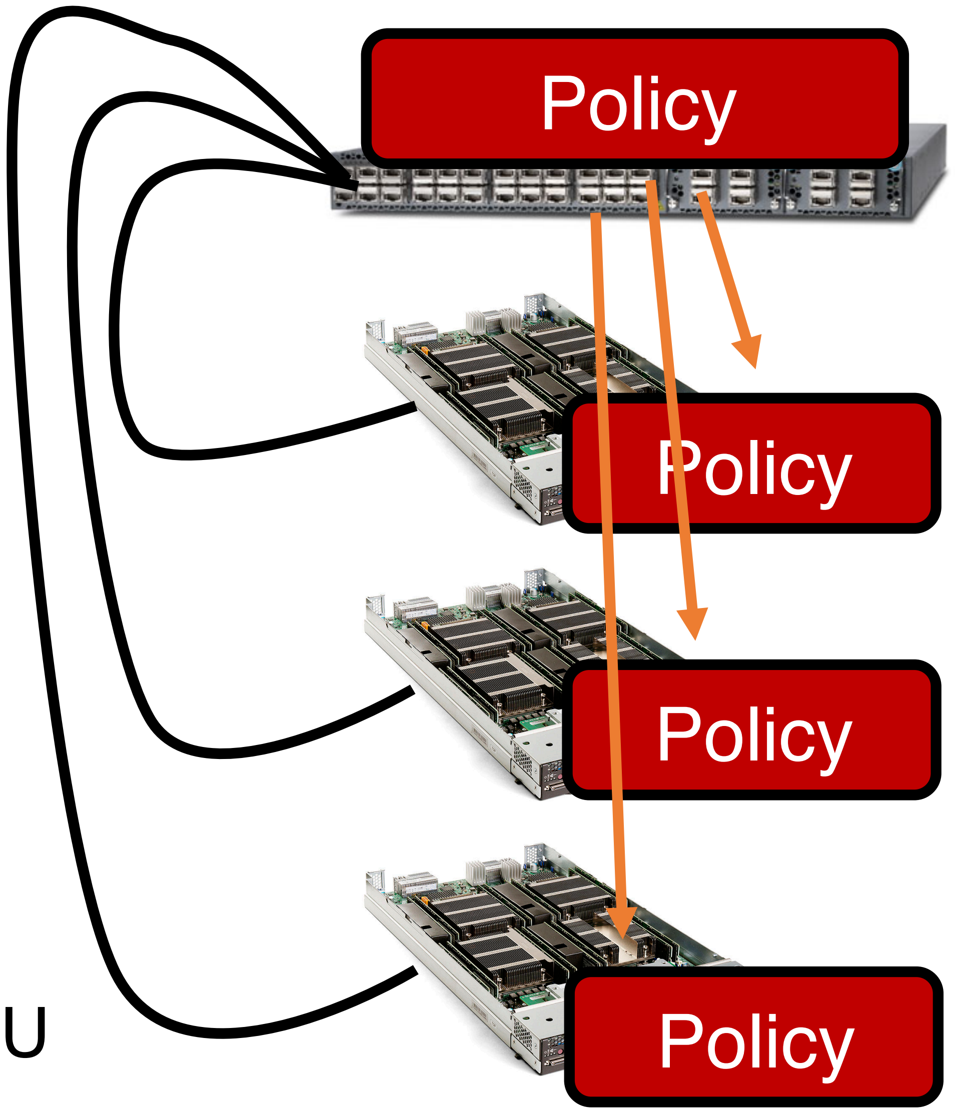
- Clusters with multiple *tenants*
- Hypervisor mediates net access
- Additional sources of overhead
  - Copy packet into VM memory
  - Simulate an interrupt to the VM
- VM continues processing packet with its own OS/network stack



Edge-core partition with virtualization

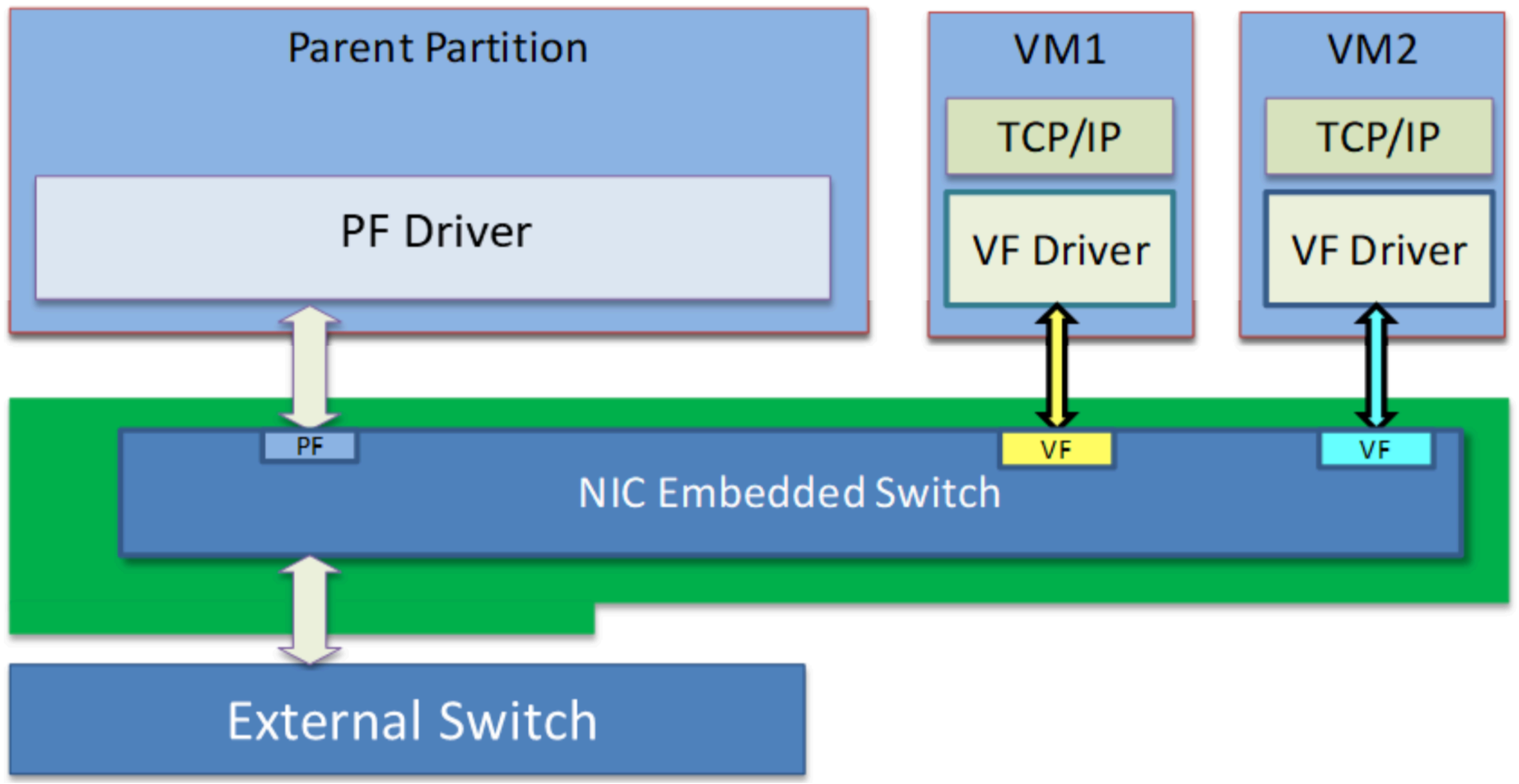
- Racks contain a bunch of servers
- Top-of-rack (ToR) switch connects
  - Servers to themselves
  - This rack to other racks
- Some new constraints…

- Complex tenant-level policies
  - Moved from core to the edge
- Use cores for apps as much as possible
  - Incentives to reduce network-related CPU



(5) Bypass the hypervisor

- Single-Root I/O Virtualization (SR-IOV)
  - Extension to PCIe bus protocol: support virtual & physical endpoints



(6) Offload to high-perf substrates

- Offloads are all about partitioning of functionality
    - “What do you run where?”
- Key idea: move fast-path processing to a faster compute substrate
    - That is, faster than x86 for the specific purpose of pkt processing ☺
- A regular NIC is an ASIC for network processing
    - High performance, low power, but not programmable
- Other ASICs: TCP offload engines (TOE), RDMA

(6) Multi-core System On Chip (SoC)

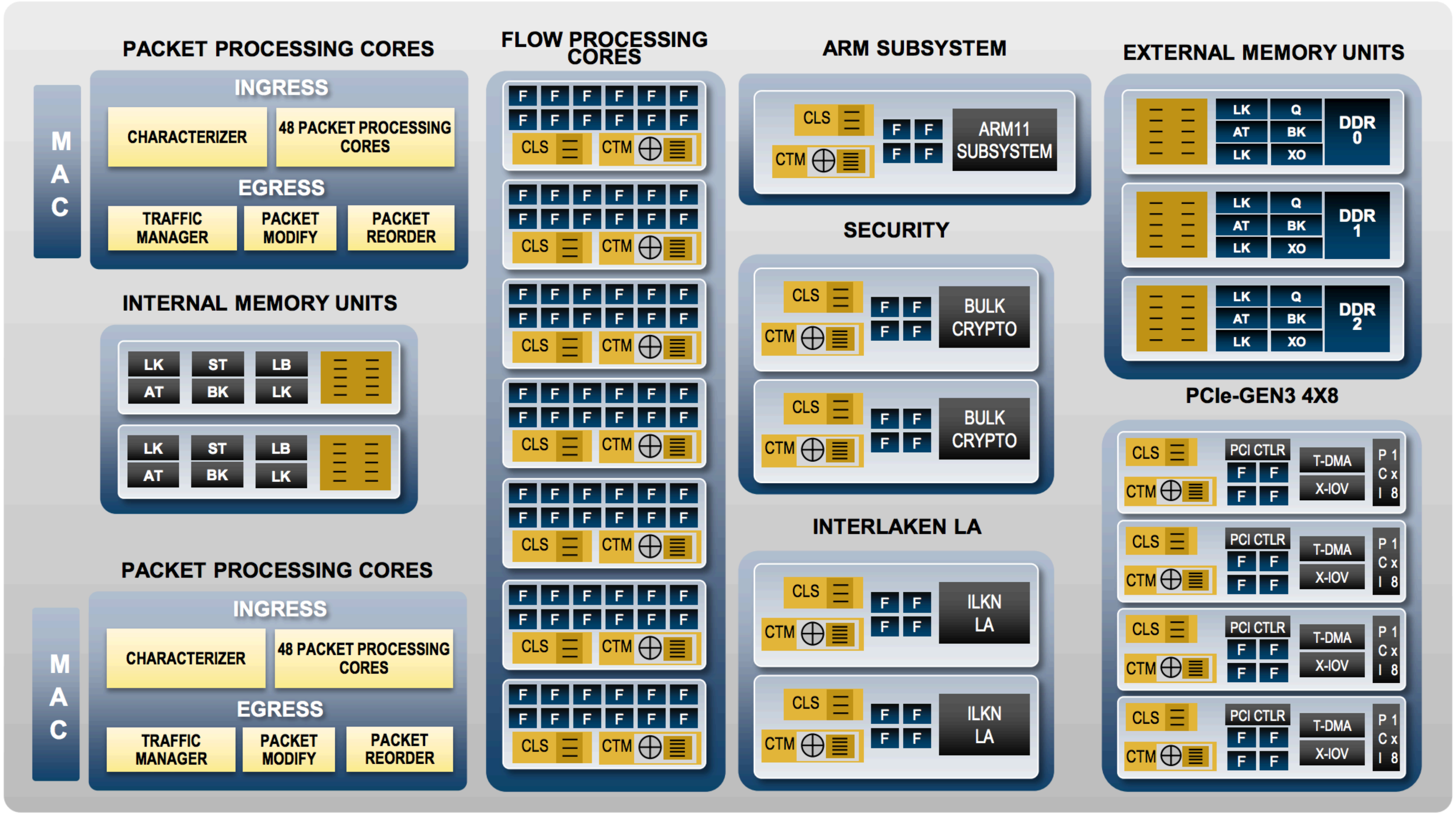
- SoC: many “small” CPU cores each of which is weaker than x86

- Many fixed-function components for specific functions
    - Cryptography, compression, hashing, etc.
    - … but general-purpose cores as flexible as x86!

(6) Multi-core System On Chip (SoC)

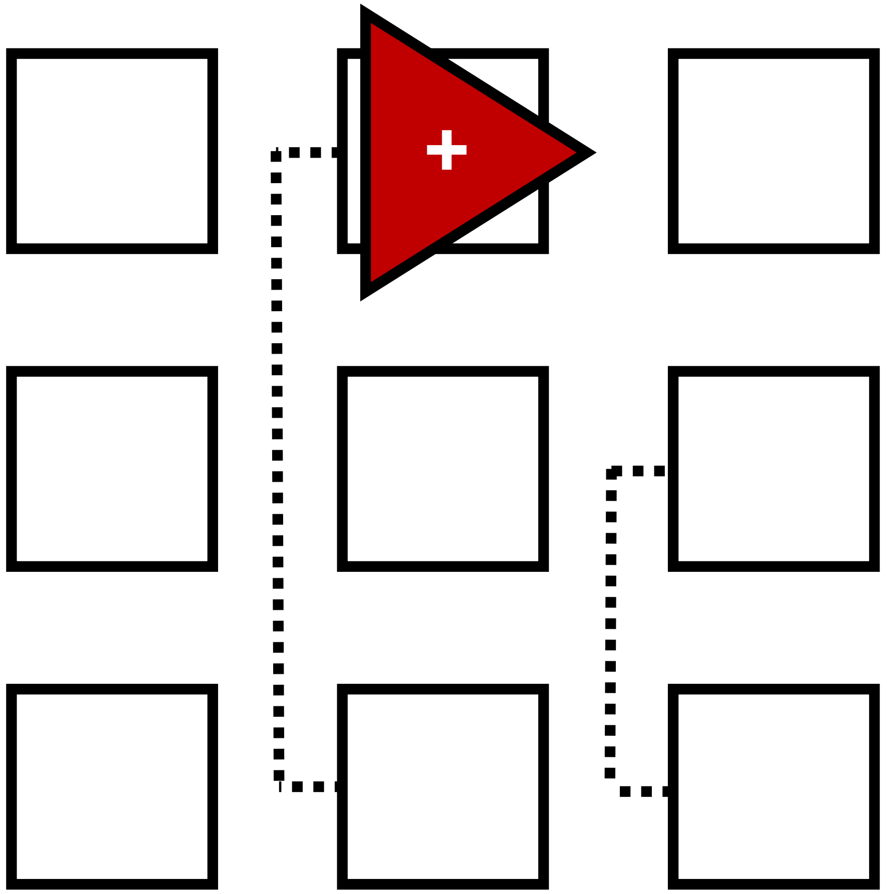
- Achieve perf by computing on packets *spatially* and *temporally*
- Spray packets across the available cores to achieve high rates
- Stateful processing: all flow’s packets must go to same core
    - Troublesome for single high-rate (ex: 100 Gbit/s) connections

(6) Example: Netronome NFP-6xxx



(6) FPGAs

- “Field Programmable Gate Arrays”
- Small programmable components called Lookup Tables (LUTs)
  - Implement any circuit of your choice
- Interconnect moves data from LUT to LUT
  - Interconnect is also configurable

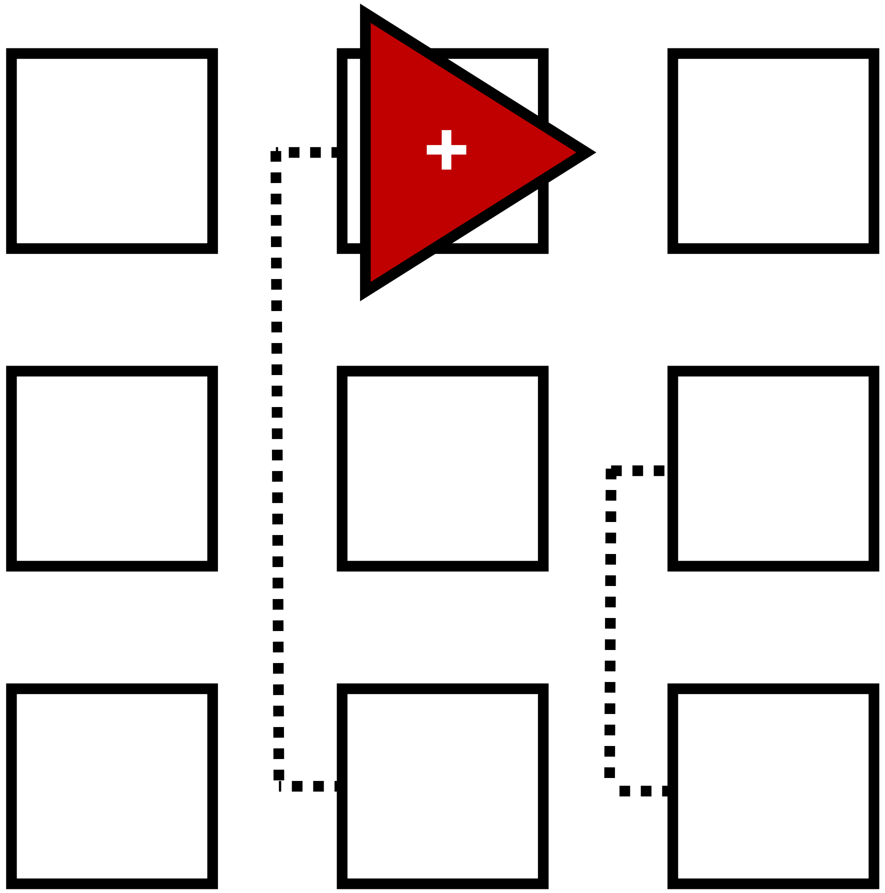


(6) FPGAs

- Achieve perf by programming functionality directly into the LUTs
  - Spatial computation
  - Thread-level parallelism
  - Pipelined parallelism

- *Usually* need circuit-level programming (ex: System Verilog) to achieve high performance

- Challenging for software developers ☹



(6) Example: Altera Arria 10

Today’s papers

- Azure NIC: offloading complex policies to FPGAs
    - Process packets without spinning CPU cores
- HotCocoa: offload congestion control to a NIC

Azure Accelerated Networking: SmartNICs in the Public Cloud

Usenix NSDI ‘18

Daniel Firestone et al.

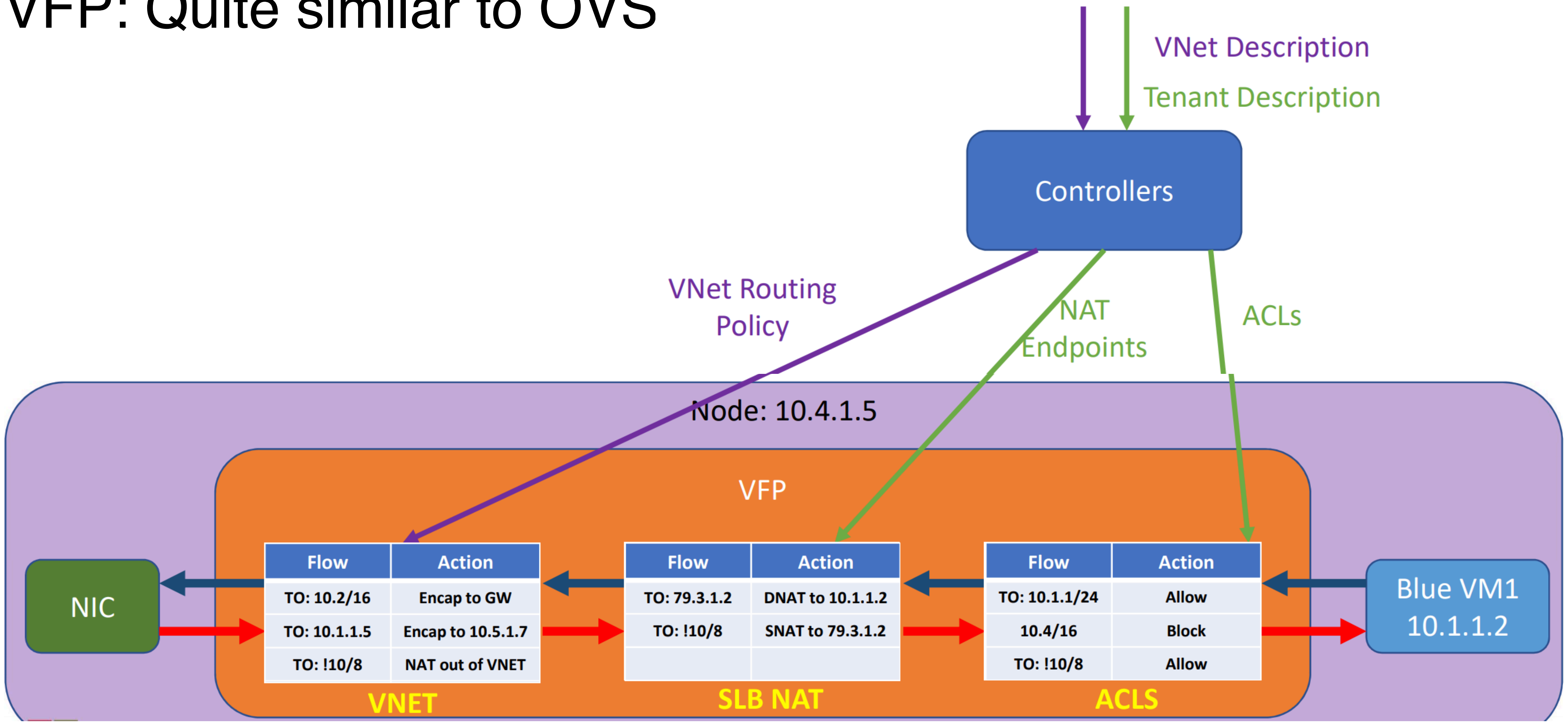
Azure host SDN

- Endpoint policies implemented on a virtual (software) switch
  - Address virtualization
  - Server load balancing
  - ACLs
  - Metering
  - Security guards

- Programmable flow tables with multiple controllers
  - Cloud provider and multiple tenants

Azure host SDN: Match-Action Tables

- VFP: Quite similar to OVS



VNET

Flow	Action
TO: 10.2/16	Encap to GW
TO: 10.1.1.5	Encap to 10.5.1.7
TO: !10/8	NAT out of VNET

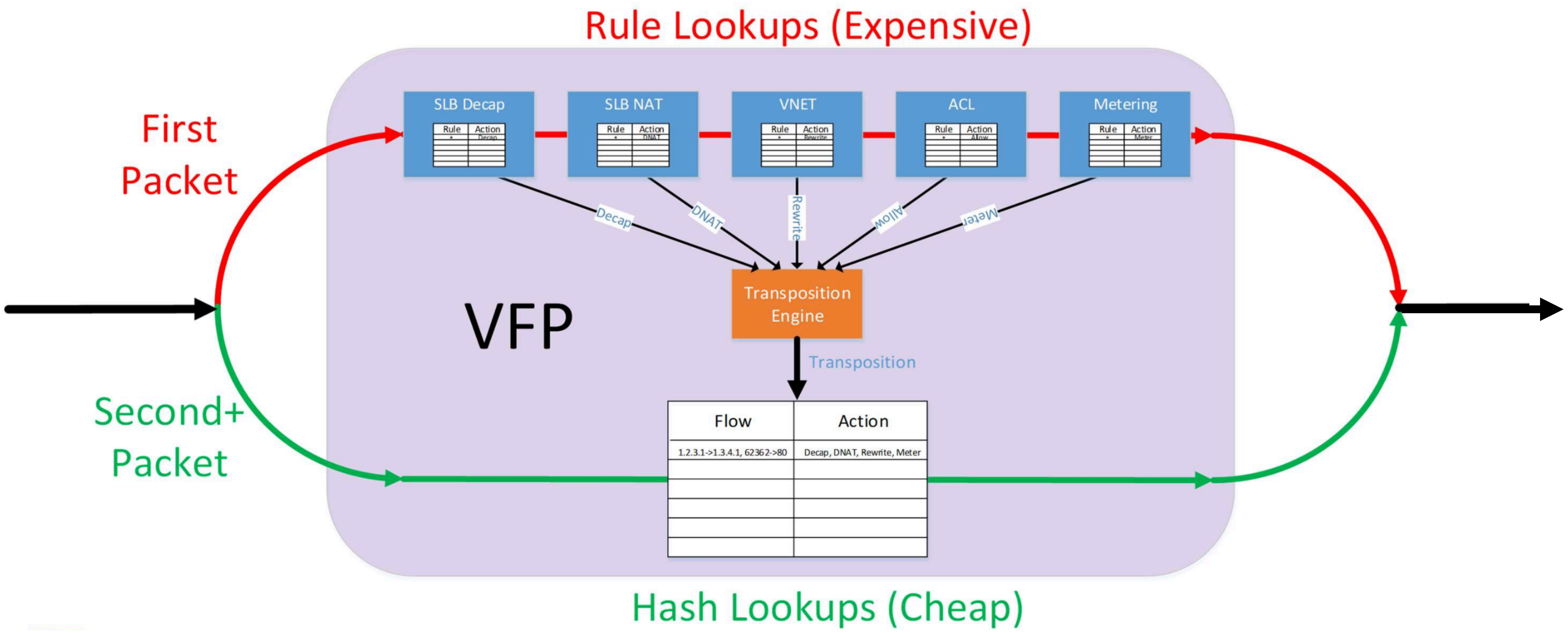
SLB NAT

Flow	Action
TO: 79.3.1.2	DNAT to 10.1.1.2
TO: !10/8	SNAT to 79.3.1.2

ACLS

Flow	Action
TO: 10.1.1/24	Allow
10.4/16	Block
TO: !10/8	Allow

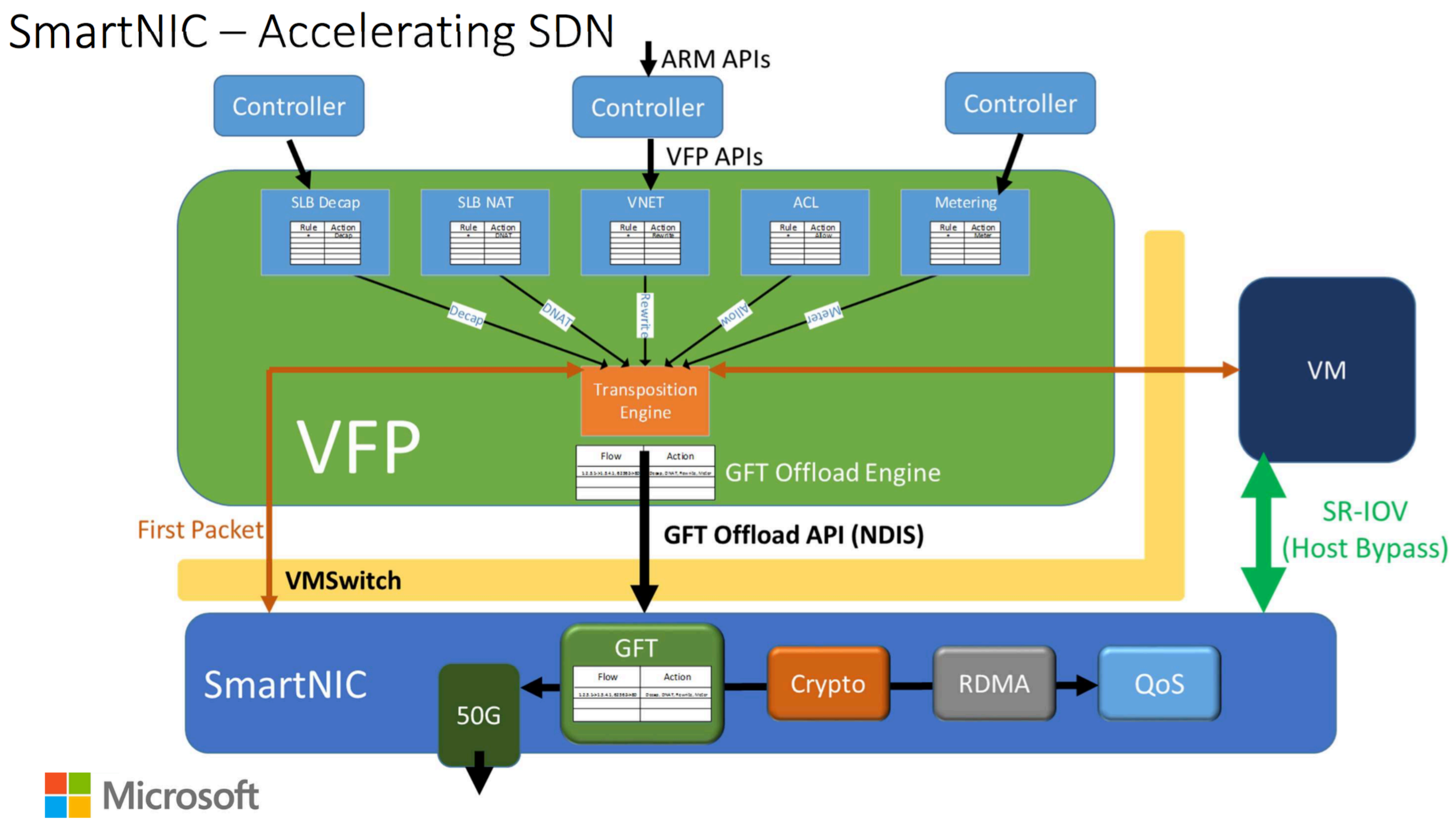
Unified flow table: A fast cache for actions



Problems with existing solutions

- Can’t scale software to 100 Gbit/s and beyond…
- SR-IOV: an “all or nothing” hardware offload
- Even if one action (e.g., TCP connection state tracking) isn’t supported, all processing must fall back to software
- *Key Q:* is it possible to achieve
    - the programmability of the host SDN software solution
    - … with the performance of hardware?

Azure NIC: an FPGA "smart NIC"



Hardest challenge: “Serviceability”

- Update parts of the system without breaking connectivity or rebooting VMs
    - … even if performance suffers (transiently)
- Includes servicing the FPGA circuitry, NIC driver, vSwitch, and the SR-IOV virtual functions
    - And, of course, updating the flow table ☺

HotCocoa: Hardware Congestion Control Abstractions

ACM HotNets '17

Mina Tahmasbi, Monia Ghobadi, Jennifer Rexford, and David Walker

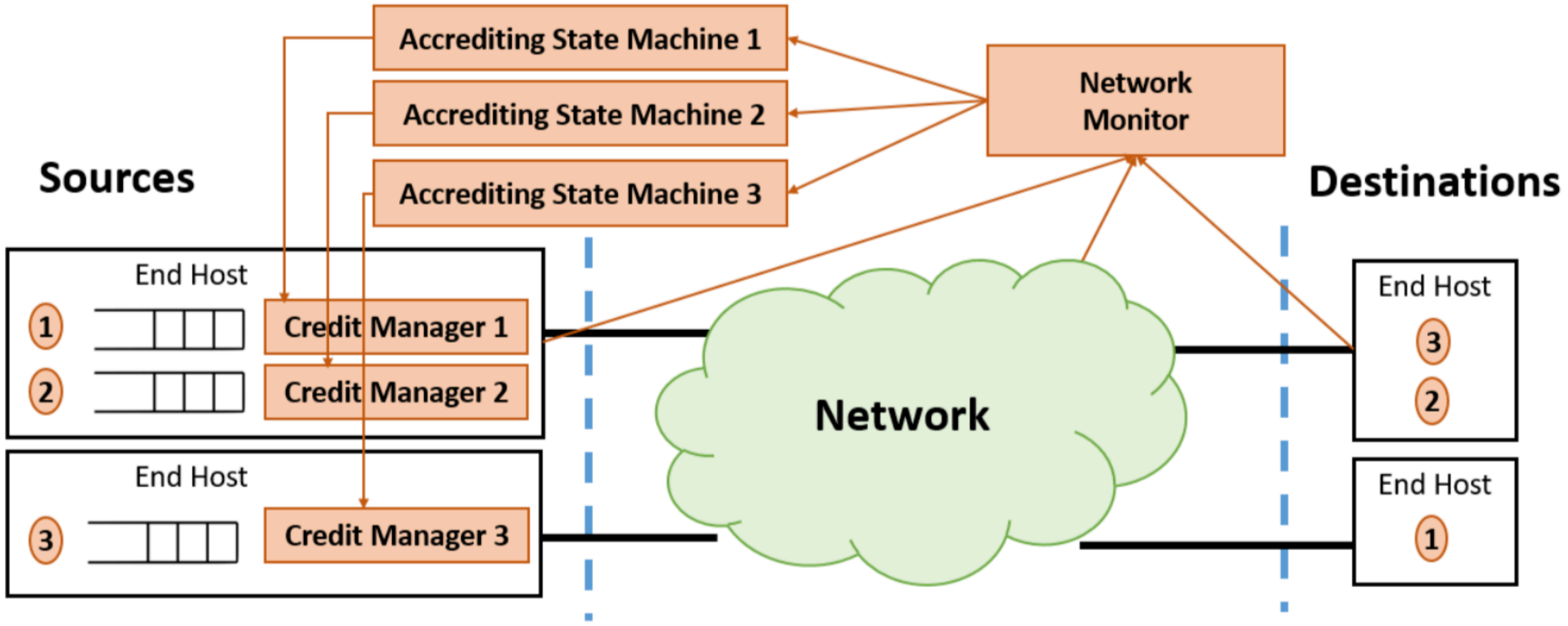
Move congestion control to NICs

- Hypervisor may retain control of the congestion state of connections in a virtualized cluster
- Example: Rate limiting
- Example: Receiver-driven flow control
- How can the hypervisor offload congestion control processing to NIC hardware
    - ... While retaining programmability to implement new schemes?

Congestion control abstractions

- Credit manager: determine when a flow may transmit packets
    - Rate-based control
    - Window-based control
    - ACK clocking
- Accrediting state machine: allocate credits based on network
    - Example: increase congestion window by one every RTT
    - State machine has “triggers” events on network conditions
- Network monitor: external entity that may trigger state transitions

HotCocoa abstractions



An example ASM

```
ASM ex_asm (Flock f, Params p) {

  State Normal{};
  State Congestion {f.cm.rate = f.rcvd_rate/f.t_rr;};

  Init Normal; // Declare initial State

  Transitions {
    | Normal on f.pkt_rcvd if f.t_rr > p.max_rate
      ==> Congestion;
    | Congestion on f.pkt_rcvd if f.t_rr <= p.max_rate
  };
}
```

Conclusions

- Programmable packet processing is an active research area
    - Lots of interest from industry and academic
- The final verdict isn’t out on the right offload substrate or partition of functions
    - The answer will depend on the use cases
- Achieving programmability with high performance may require deep expertise in different areas
    - Processor and interconnect architectures
    - Compilers
    - Packet-compute scheduling

Backup Slides

Outline (2/2)

- Substrates for offload: ASIC, multicore SoCs, FPGAs, no offload
- Azure NIC
  - Virtual filtering platform offload: needs?
  - Control-data plane separation
  - FPGA pipeline design
  - Future work: QoS. Better scheduling. Better monitoring
- HotCocoa
  - Congestion control at the hypervisor without the overheads
  - credit management (e.g., window vs. rate) – data plane
  - accrediting state machine (ASM) – control plane