Network
Hardware Router overview

- Input port
- Input port
- ... Input port
- Switching fabric
- Output port
- Output port
- ... Output port

- Line Termination
- Parsing
- Lookup & Modification
- Buffering & Scheduling
- Line Termination

(Match-Action) (Traffic Management)
Nonblocking designs are nontrivial

Two aspects: topology and routing
3-stage Clos network (r*n X r*n ports)
Rearrangeably nonblocking Clos built with identical switches: $n^2 \times n^2$ using $3n \times n$
Increasing # ports: Butterfly Networks

• Can we reduce internal # ports for a given external # ports?

• K-ary L-butterfly:

• Use L*K KxK port switches to build K^L-XK^L port switch

• Figure: K = 2, L = 3

• Produce n^3Xn^3 switch from 3n nXn switches
  • Clos: n^2Xn^2

• Routing is deterministic

• Tradeoff: more blocking

https://ece757.ece.wisc.edu/lect09-interconnects-2-topology.pdf
(4) MGR: Crossbars & Matching

- MGR uses a nonblocking crossbar across 15 ports

- Strategies to match incoming demands & output ports quickly
  - Greedy (simple), wavefront, group
  - Try to address fairness across ports
4) RMT: Memory switching fabric

- RMT uses memory as the fabric to hold packet headers and payloads between any two interfaces.
- Key challenge: simultaneous access to memory (N memory ports).
- In the late 90s and early 2000s, there was considerable research on building high-speed packet buffers.
- Today: shared memory switches & routers (shared across ports).
  - Fast memory can be clocked at 1 GHz.
- Fundamental tradeoff: faster memories are not very dense.
  - Can’t make the memory too large; can’t hold too many packets.
- Workaround: exploit memory access patterns: e.g., each queue is FIFO.
- Traffic manager implements scheduling & buffer management.
(5) Traffic Manager

- Where should the packets not currently serviced wait?
- Two designs: Input-queued vs. output-queued
- Output queueing avoids **HOL blocking** exhibited by input queueing.
  - Suppose port 1 wants to send to both 2 and 3 but port 2 busy
  - Packets from p1 towards p3 need not be delayed
(5) Traffic Manager

• Queueing represents output port contention
• A single output port can be represented by multiple queues
  • e.g., to implement weighted fair queueing
• Each queue is just a linked list in the shared memory
  • Maximum flexibility in queue sizes, but pointer overhead
  • Separate memory to maintain per-queue heads and tails
(5) Traffic Manager: Scheduling policies

• How to dequeue packets in output port buffer? packet scheduling algorithms
• Fair queueing across ports or flows
• Strict prioritization of some ports over others
• Rate limiting per port
• Possible to make it flexible: PIFOs
(5) Traffic Manager: Buffer Management

- Q: how to enqueue packets into buffer?
  - If buffer is full, which packet should be dropped?

- Typical buffer management: Tail-drop

- Want fairness: if queue 1 has too many buffered pkts, don’t tail-drop q2
  - Share memory by partitioning (carving memory out) across queues

- Want efficiency: if q1 has no pkts, q2 should be able to use (nearly) all buffer memory

- One possibility: static thresholds for buffer occupancy per port
  - Can be made fair or efficient but not both
(5) Demand-aware buffer management

- DT: “Dynamic Queue Length Thresholds for Shared-Memory Packet Switches”, Choudhury and Hahne
- Compute a critical (dynamic) queue length threshold $T$

$$T(t) = \alpha \cdot (B - Q(t)) = \alpha \cdot \left( B - \sum_i Q^i(t) \right)$$

- Port blocked from adding packets if

$$Q^i(t) \geq T(t)$$
Egress line termination

- Combine headers with payload for transmission
  - Must incorporate effect of header modifications
  - Also called deparsing or serialization

- Multicast: egress-specific packet processing
  - Ex: different source MAC address for each output port

- Multicast makes almost everything inside the switch (interconnect, lookups, queueing) more complex
Note: three kinds of router hardware data plane programmability

• Packet header formats, i.e., the packet parser
  • Example: Go from IPv4 -> IPv6
  • Custom packet format to carry financial info at high speed on a point-to-point link

• Table formats, actions, sizes, i.e., the match-action tables
  • Change which fields in the packet can be processed by a table
  • Control the table sizes, i.e., # entries, and hence the memory resource footprint according to use case.
Note: three kinds of router hardware data plane programmability

- Packet scheduling, i.e., the traffic manager
  - Flexible classification of packets
  - Flexible assignment of ordering and timing of when packets are transmitted from an outgoing link
… which is distinct from control plane programmability

- The control plane must compute the packet-processing rules put into the memory on the router ASIC
  - Example: packet with IPv4 destination 10.0.0.1 must go out of port 4

- Data plane programmability refers to the flexibility in the allowed set of packet headers, tables, and actions themselves, not the actual rules.
  - Example: There is a table that matches on IPv4 destination addr whose action is to determine the output port
Software Data Plane
Why software?

• Applications run in software. Get packets to/from apps quickly

• Software routers:
  • virtualization and cloud (e.g., openvSwitch)

• Middleboxes (network functions)
  • Network Address Translation, mobile processing nodes (packet gateways, radio controllers, …), tunneling gateways (IPsec/SSL VPN), traffic analysis & security (IDS, firewalls, spam), CDNs/caches, video accelerators, …
Packet processing on Linux

Receive path
How is data received in software?

• Have CPU poll the network interface card (NIC) memory to copy data

• Interrupt from the NIC (“data is available”), then CPU reads memory

• Direct Memory Access (DMA): NIC moves data to memory
  • Reduce or remove CPU from the “data moving” loop
  • Large data or scattered data
Modern NICs and architectures can also do direct cache access (DCA)
Revisiting network I/O APIs: The netmap framework. CACM’12
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
Allocate packet data structures in memory (sk_buff, mbufs, …)

Optionally, steer packet to core running the application
Socket buffers

- Allocate in arbitrary chunks (multiples of 64 bytes)
- Support arbitrary packet sizes, fragments, deferred processing
Other things that happen afterward

- Netfilter: tracking TCP connection state, firewalling, NAT, …
- Packet scheduling decisions
- IP protocol processing: routing
- Transport processing (UDP/TCP protocol layer)
- Copy into user space socket buffers

- Some stateless, per-packet work can be done by the NIC:
  - TSO: TCP segmentation offload
  - LRO: Large Receive Offload
  - IP checksum (transmit & receive)
FreeBSD `sendto()` code path

Overheads are sprinkled throughout the packet processing stack.

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

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
(3) Amortize operations: batching

- Notifications to NIC for packets written for transmission or free buffers available for reception

```c
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) {

        /* Send burst of RX packets, from first port of pair. */
        struct rte_mbuf *bufs[BURST_SIZE];
        const uint16_t nb_rxe = rte_eth_rx_burst(port, 0,
                                                bufs, BURST_SIZE);
        if (unlikely(nb_rxe == 0))
            continue;

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

        /* Free any unsent packets. */
        if (unlikely(nb_txe < nb_rxe)) {
            uint16_t buf;
            for (buf = nb_txe; buf < nb_rxe; buf++)
                rte_pktmbuf_free(bufs[buf]);
        }
    }
}
```
The abstraction has changed!

• Fast packet processing frameworks (netmap, DPDK, eBPF) move data to application buffers very 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
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!
Policies in virtualized switches

• Tenant policies
  - **Network virtualization:** I want the physical network to look like my own, and nobody else is on it

• 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?

Hypervisor (OR) orchestrator
OpenVSwitch: Requirements

• Support large and complex policies

• Support updates in such policies
  • Q: why?

• 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
  - ... (Ellipsis for VM n)

- Hypervisor
  - ovsdb-server
  - ovs-vswitchd
  - kernel module

- Controller
  - OVSDB
  - OpenFlow

- NICs
  - Netlink

- User
  - Kernel
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 algos
Idea 1: Microflow cache

- Microflow: complete set of packet headers with action
  - Example: srcIP, dstIP, IP TTL, srcMAC, dstMAC
- Use tuple space search to do one 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

- 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

Use tuple space search

Hit

Miss

Megaflow cache in the kernel

Openflow table in user space
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: application must be modified
  - Device drivers must often be modified
- Multitenancy: think about implications to weakening fault isolation
- Can we get isolation with efficiency?
Additional issues to consider

• Safe & efficient composition of middleboxes

• Placement and routing

• “Expressiveness” of your application: floating point, vector, …