OVS: accelerating the datapath through netmap/VALE

Luigi Rizzo
Universita` di Pisa, Italy

(SW) Data plane performance

Depends on many components

- basic I/O costs (hw or virtual)
- flow table performance
- packet size
- traffic patterns
- hypervisor's datapath

Talk overview

- background on netmap/VALE
- integration with OVS (user and kernel dataplanes)
- hypervisor and guest datapaths
- future work

Device I/O

Expensive even on bare metal

- at least with standard device drivers
- DPDK, netmap, packetshader, PF_RING/DNA show that we can go fast
- netmap shows we only need minimal driver changes

What are the options for virtual ports and VMs ?

- tap is slow (one packet per transaction)
- shared memory for efficient data transfer (protection ?)
- batching for efficient signaling

Netmap goals and history

Build a fast path between NIC and applications

- targeted to raw packet I/O
- robust, easy to use, device independent
- use the OS for what it is good at

Evolution

- jun. 2011: first prototype and FreeBSD release
- feb. 2012: linux release
- jun. 2012: VALE (virtual software switch)
- jan. 2013: Qemu extensions
- dec. 2013: netmap pipes
- apr. 2014: bhyve support

Netmap + VALE + netmap pipes

Core netmap code available at

code.google.com/p/netmap

- in-tree for FreeBSD 9, 10 and HEAD
- out-of-tree kernel module for linux 2.6.32 and above

Applications (both from us and from third parties):

- netmap-ipfw (userspace ipfw/dummynet)
- netmap-libpcap (-> usable by libpcap apps)
- netmap-click (-> usable by Click apps)
- qemu, Xen, bhyve support

Public repositories at

code.google.com/p/netmap-*

Netmap design principles

Key problem:

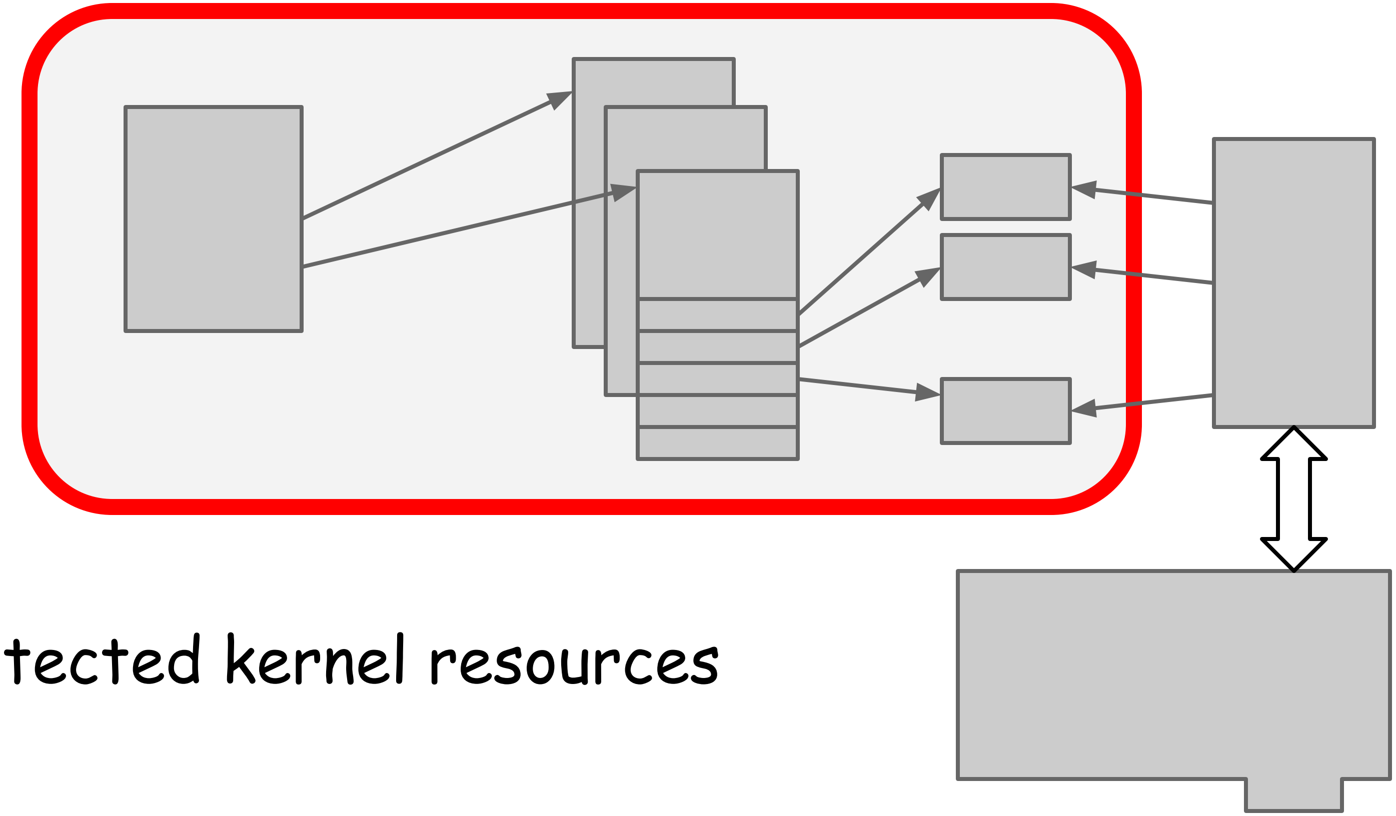
cut down per-packet processing costs

Amortize -> batching
Remove -> preallocation, mmap
Reduce -> one flat packet format

Data structures

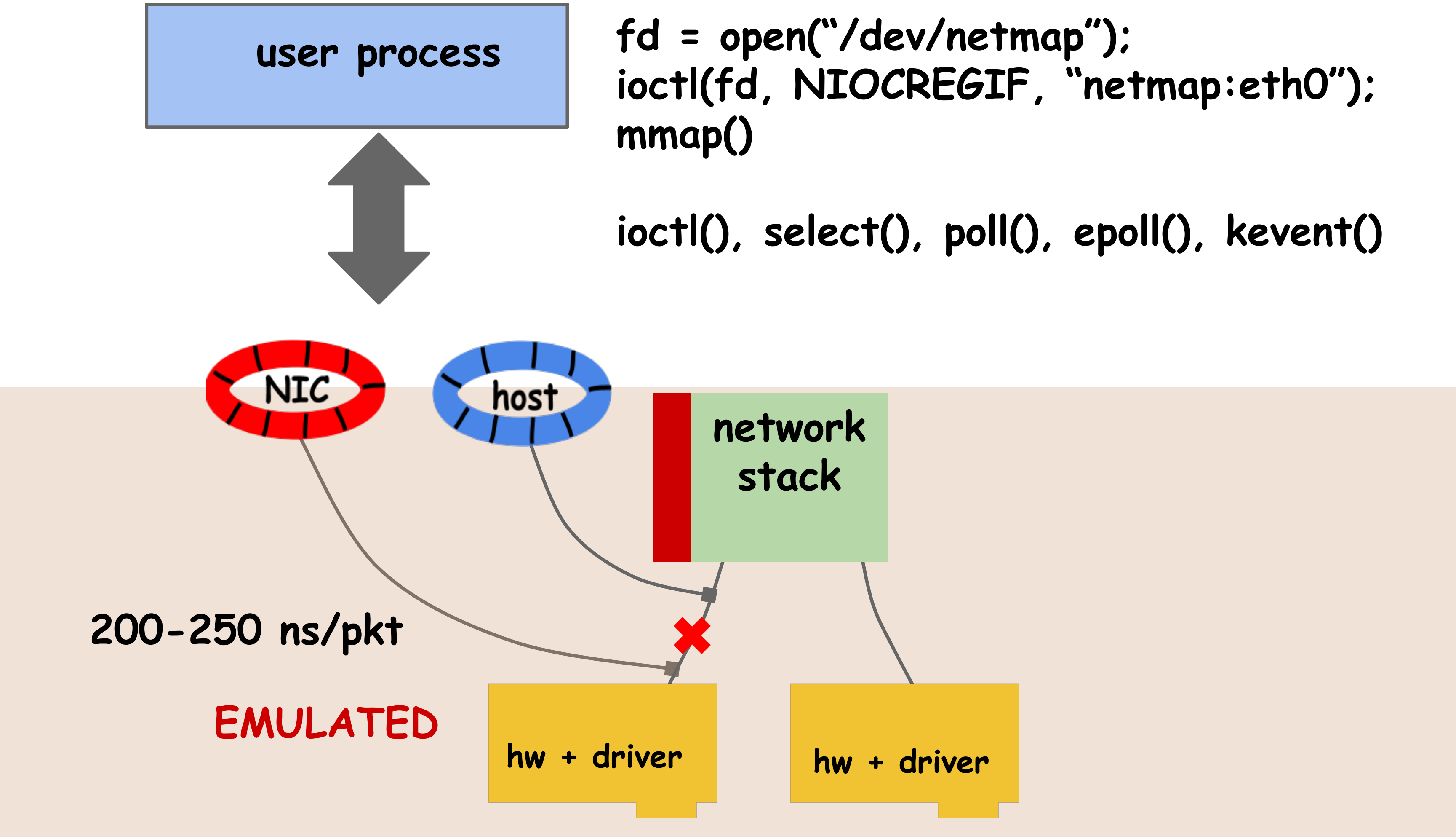
(Similar for netmap, dpdk, PF_RING/DNA)

shared data structures: netmap port



protected kernel resources

Netmap NIC access



user process

```
fd = open("/dev/netmap");
ioctl(fd, NIOCREGIF, "netmap:eth0");
mmap()

ioctl(), select(), poll(), epoll(), kevent()
```

NIC

host

network stack

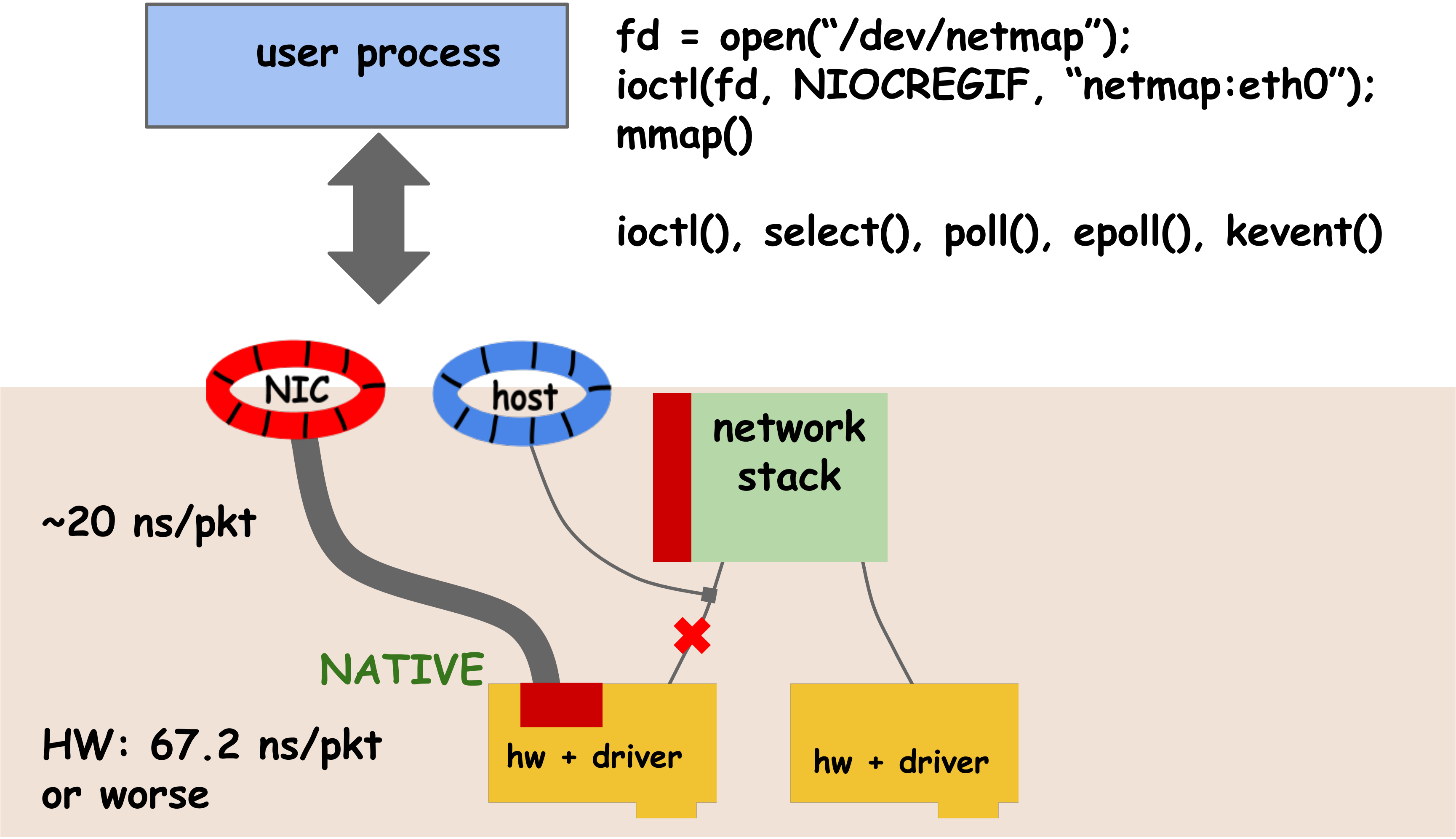
200-250 ns/pkt

EMULATED

hw + driver

hw + driver

Native NIC access



```
fd = open("/dev/netmap");
ioctl(fd, NIOCREGIF, "netmap:eth0");
mmap()
```

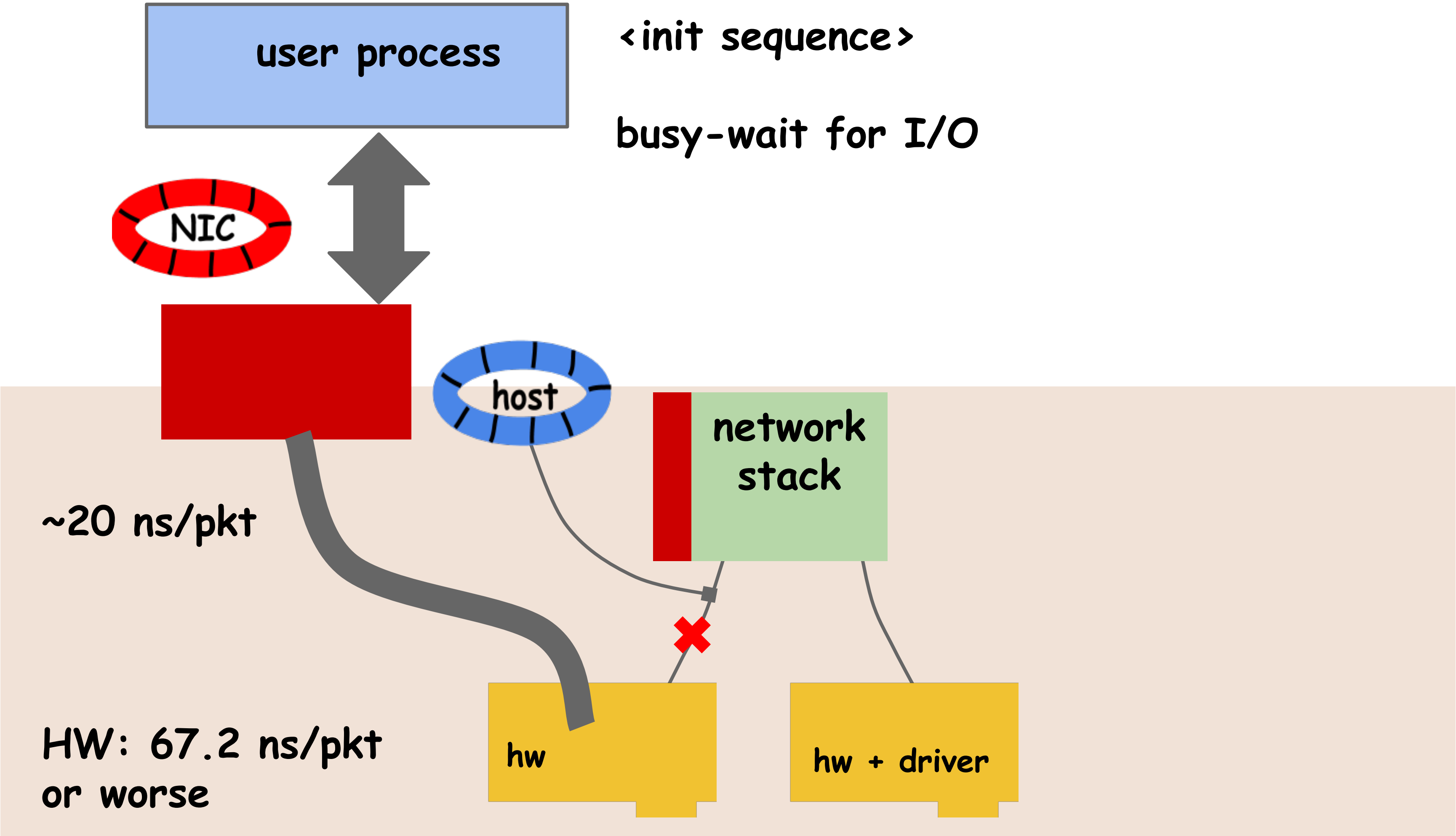
ioctl(), select(), poll(), epoll(), kevent()

~20 ns/pkt

HW: 67.2 ns/pkt
or worse

http://info.iet.unipi.it/~luigi/

DPDK NIC access



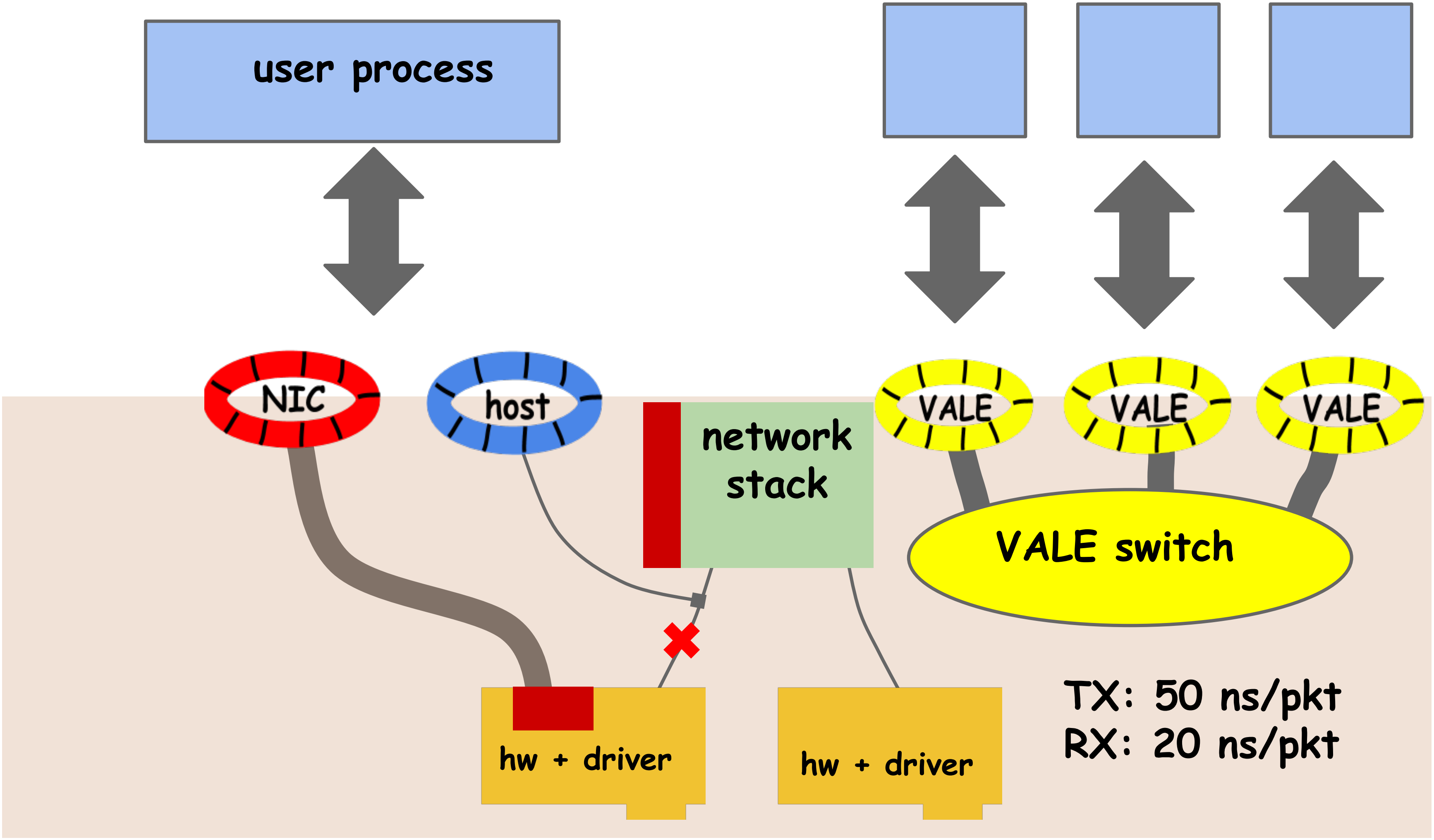
<init sequence>

busy-wait for I/O

~20 ns/pkt

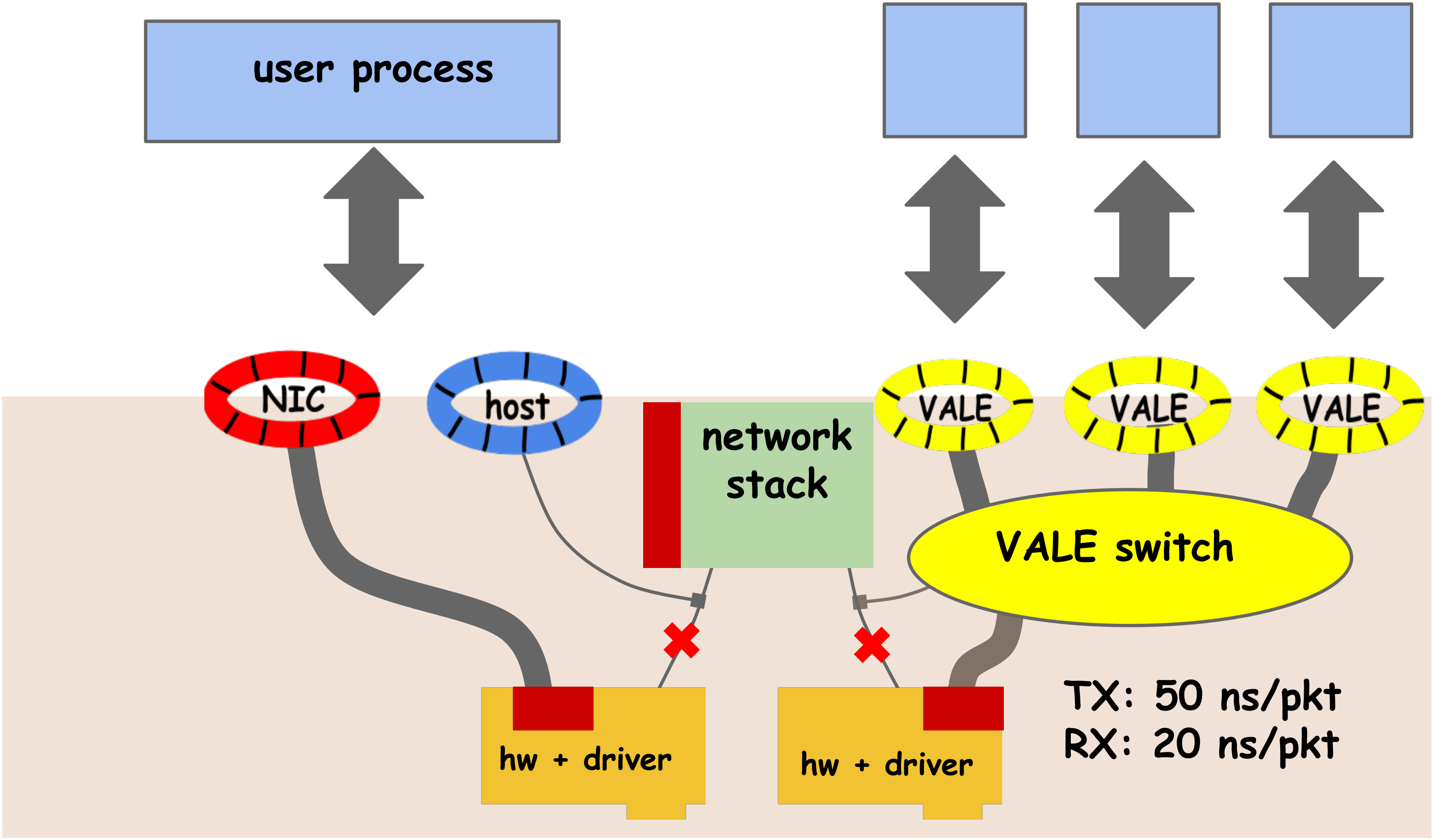
HW: 67.2 ns/pkt
or worse

VALE switch



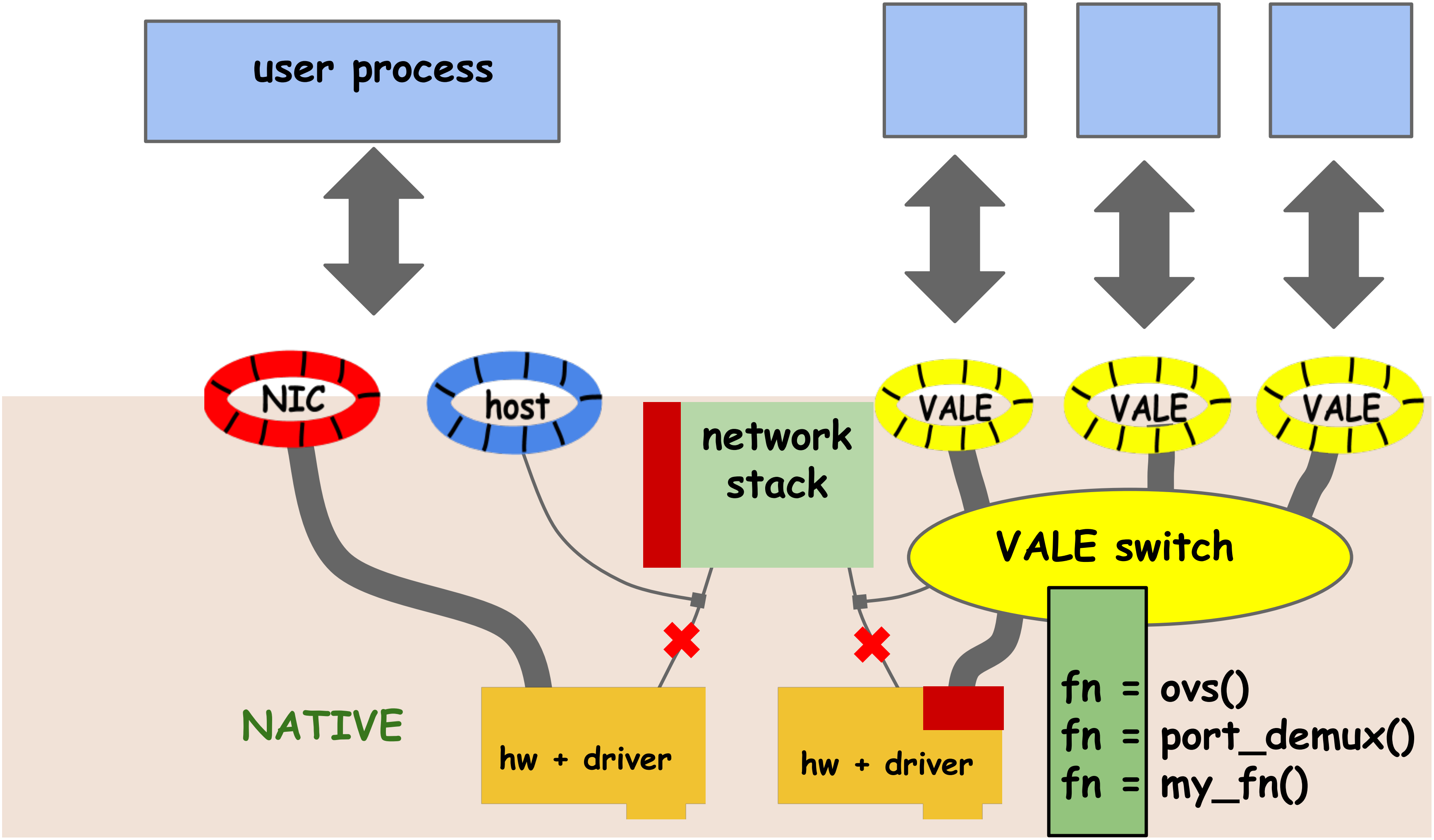
TX: 50 ns/pkt
RX: 20 ns/pkt

VALE switch + NIC/host

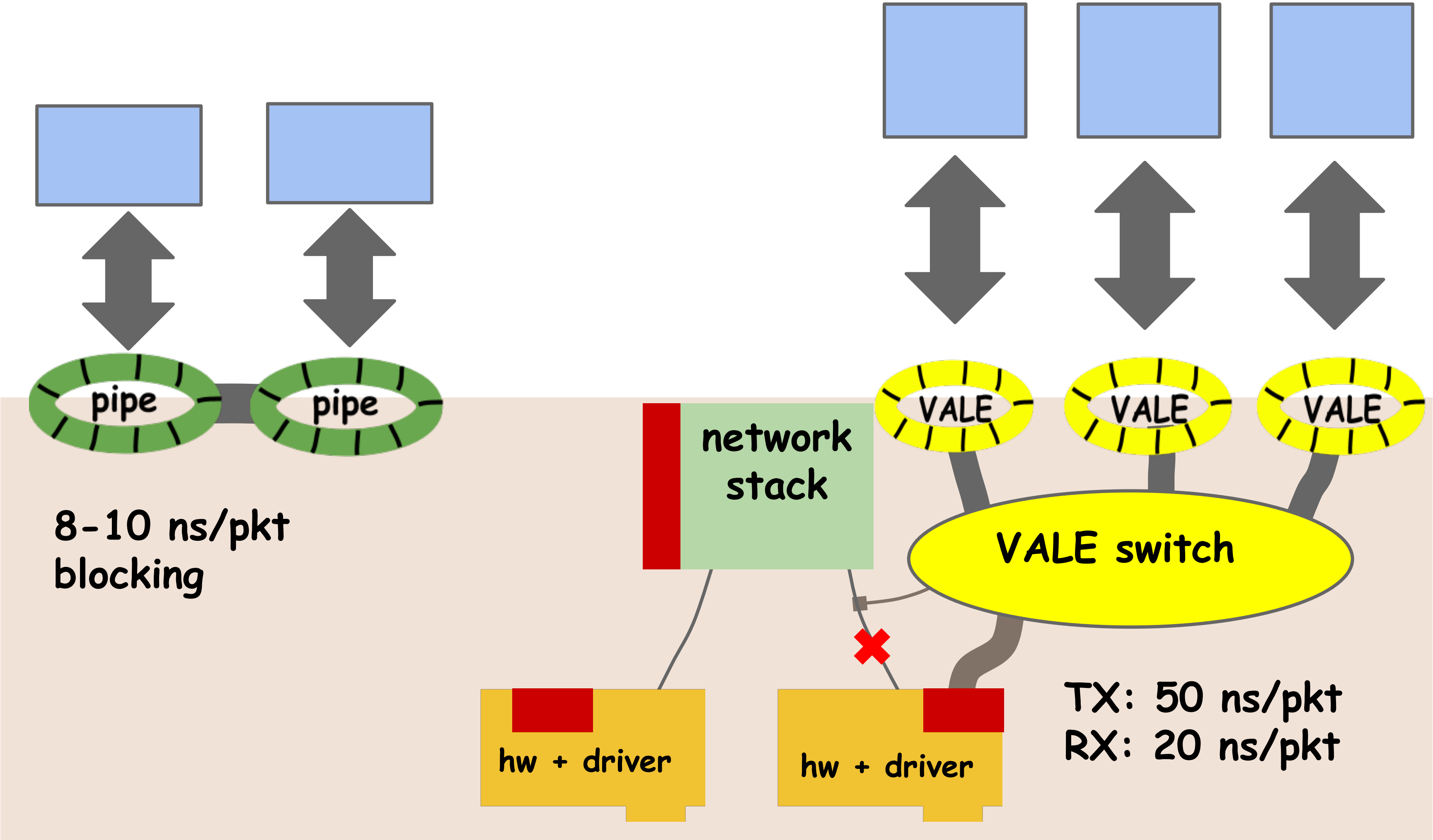


TX: 50 ns/pkt
RX: 20 ns/pkt

Custom logic, VALE dataplane



Netmap pipes



8-10 ns/pkt
blocking

TX: 50 ns/pkt
RX: 20 ns/pkt

Performance

(single core, best case, large batches, aligned packets, ...)

Basic I/O (netmap in OR out, device): 20 ns/pkt

- many NICs cannot do line rate due to their own hw limitations
- PCIe bus accesses also problematic with strange lengths or unaligned packets

VALE switch (one data copy) 50..250 ns/pkt

- 20 Mpps 64 bytes, 4 Mpps/50 Gbit/s 1500 bytes
- scales to memory bandwidth with multiple senders

netmap pipes (zero copy) 8-10 ns/pkt

- mostly insensitive to packet size

OVS and netmap: userspace

Userspace datapath (2011)

- create PCAP port type for the userspace datapath
- add an extra thread for the event loop
- exploit batching
- use pcap-over-netmap for I/O

Throughput up to ~3 Mpps (NIC to NIC)

OVS and netmap: kernel

In-kernel datapath (2013)

- use VALE as as a dataplane
- replace the lookup function with

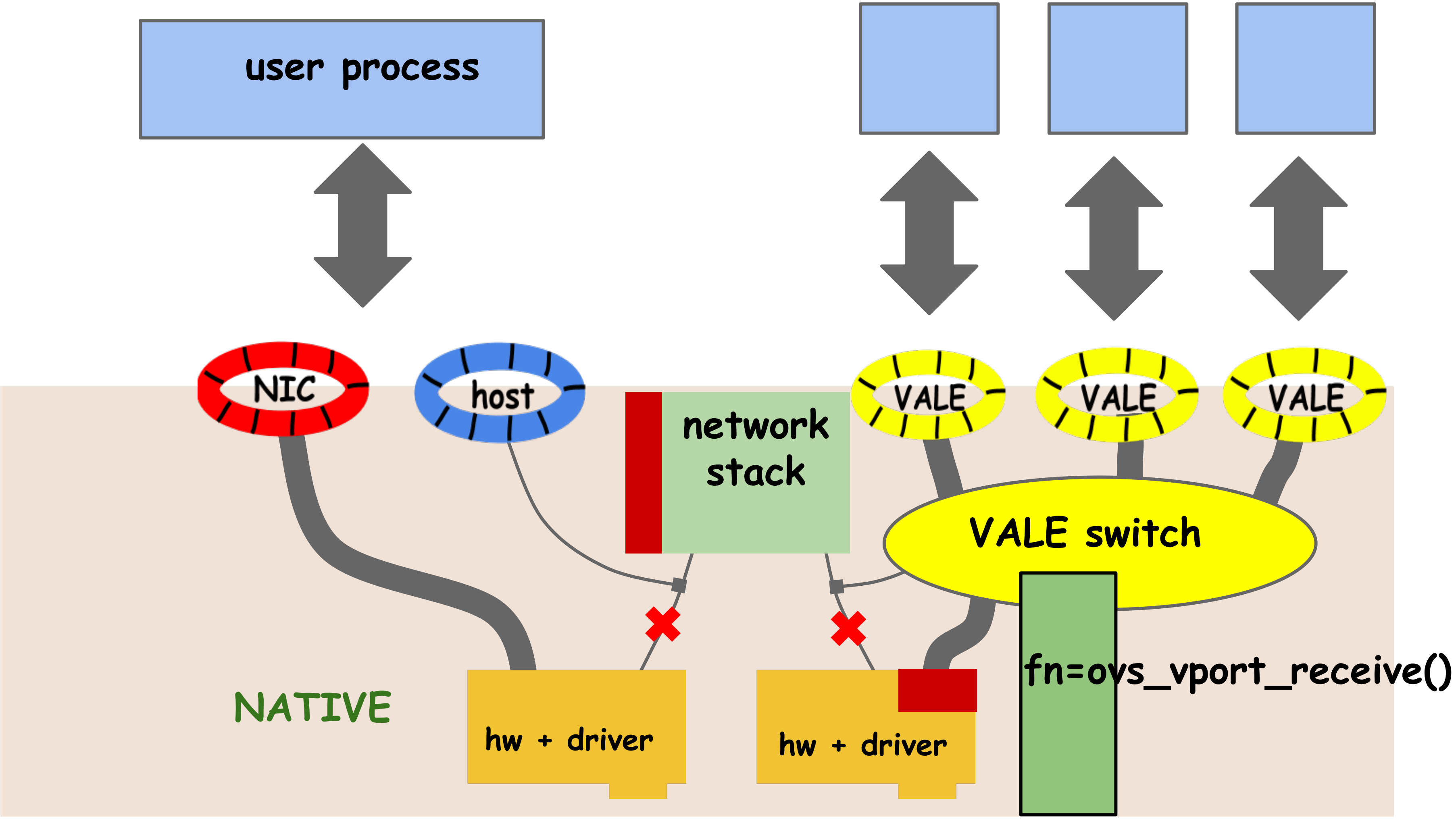
  ovs_vport_receive(vport, skb);

Throughput up to 3 Mpps (NIC to NIC)

Much room improvement:

- reduce wrapping costs
- batching in ovs_dp_process_received_packet()

OVS logic, VALE dataplane



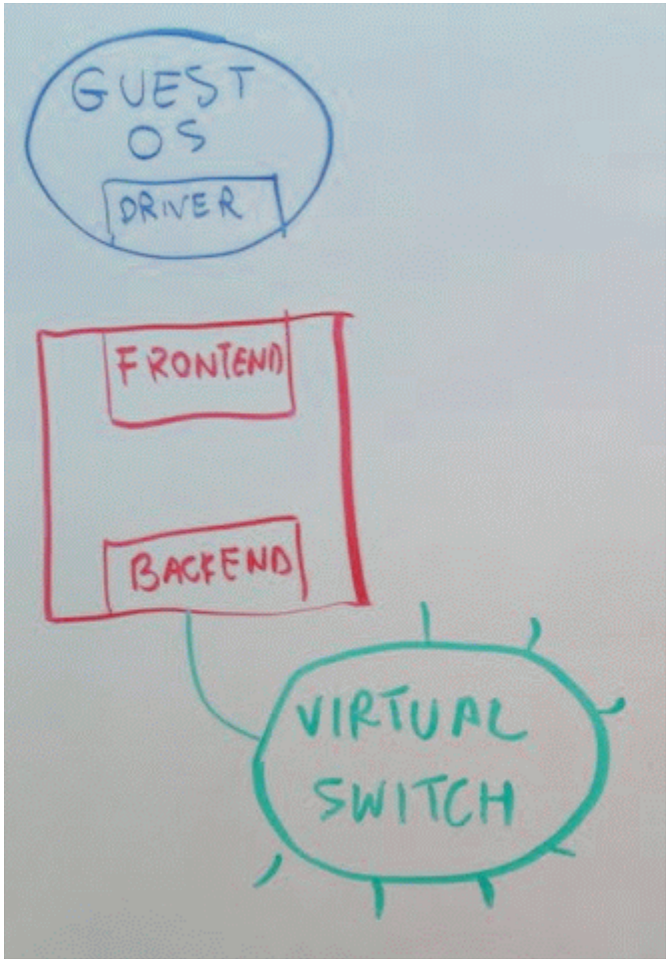
Network datapath for VMs

Speed depends on the slowest component

- paravirtualized drivers (even a simple e1000 is fast)
- frontend/backend speedups
- replace tap with faster APIs

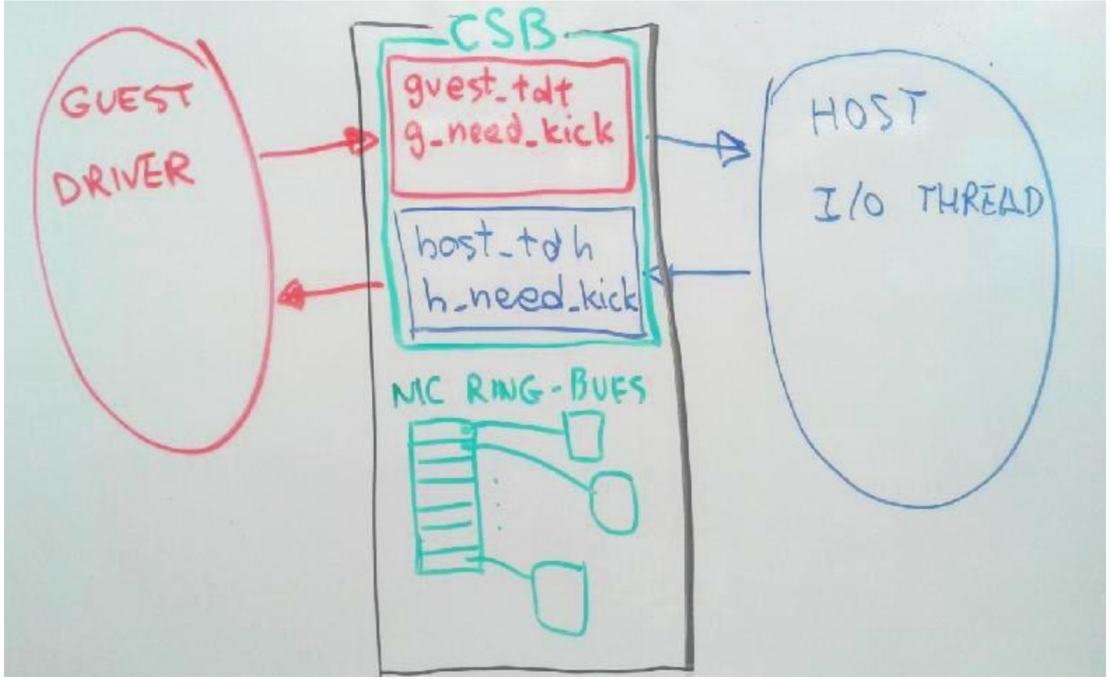
Common tricks:

- batching ("more flag", 2012)
- amortize exits
- fewer copies



Paravirtualized drivers

- notifications sent through shared memory
- host thread polls device status
- mechanism to start/stop polling threads



Paravirtualized drivers

How useful is the I/O thread ?

reduce exits

- large speedups (2-5x)
- can be done in other ways:
  send-combining, interrupt moderation
  (non-pv e1000 with moderation +SC as fast as virtio)

remove exits

- up to an additional 2x speedup
- requires matching speed in producer and consumer

guest app ↔ frontend ↔ backend ↔ switch ↔ ...

Batching is key for performance

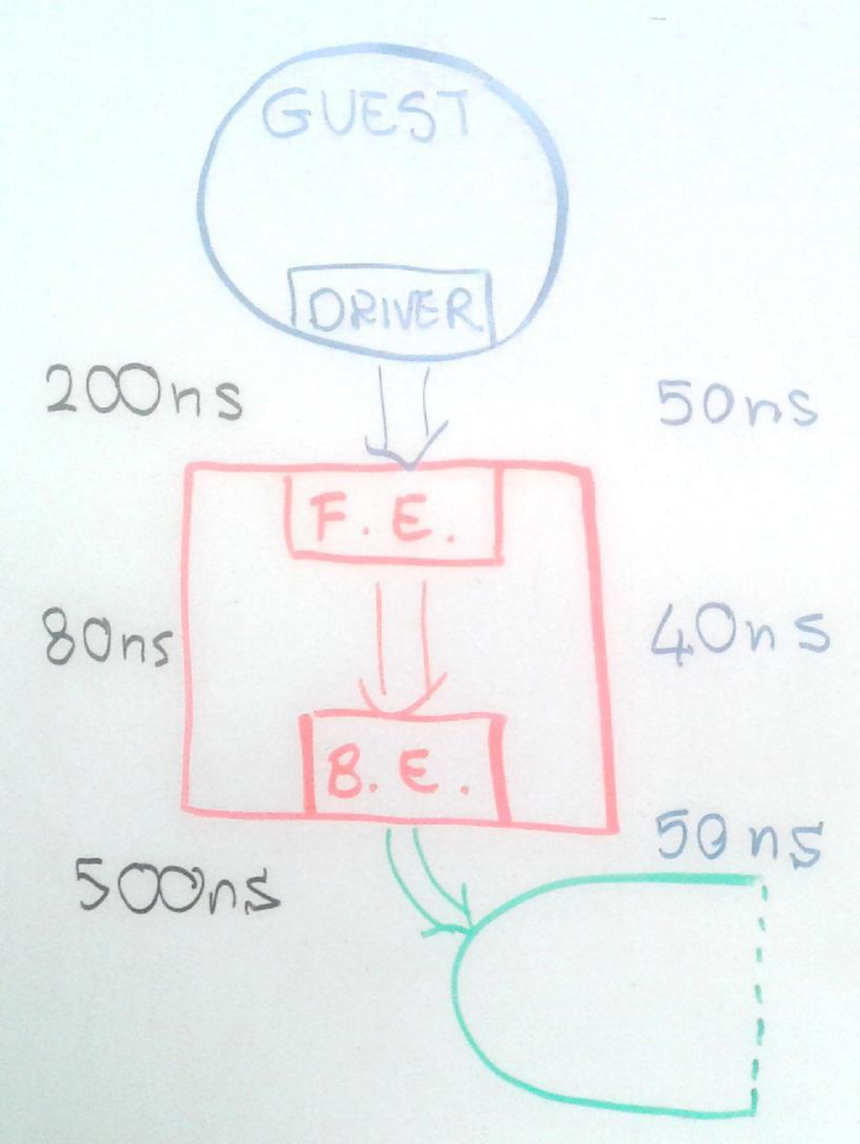
If possible, extend APIs to support batching

- reasonably feasible in the input path

otherwise, infer from actual traffic

- use pending interrupts or signals as hints (send combining)
- in late 2012 we proposed a qemu flag, QEMU_NET_PACKET_FLAG_MORE

Example: qemu



Hypervisor netmap support

(single core, best case, large batches, aligned packets, ...)

QEMU: up to 6-8 Mpps G-G, 12 Mpps G-H

- basic netmap support in-tree (3-4 Mpps)
- more flag, PV netmap in guest, indirect buffers not committed yet

bhyve: ~8 Mpps G-H

- full netmap and virtio support

Xen: 6-10 Mpps G-H

- first approach, replace xen rings with VALE
- current approach: netmap extension for netfront/netback
- use VALE in DOM0

Future work

Passthrough mode for netmap

- zero overhead data transfer
- signalling still goes through the hypervisor

Encapsulations and offloadings

- TSO already available
- others will be added as needed

Upstreaming

Acknowledgements

Funding and support (over time):

Intel Research, EU FP7 (Change, Openlab), ACM, Netapp, NEC, Cisco, Verisign

Developers:

Luigi Rizzo, Giuseppe Lettieri, Michio Honda, Matteo Landi, Gaetano Catalli, Vincenzo Maffione, Stefano Garzarella, Joao Martins