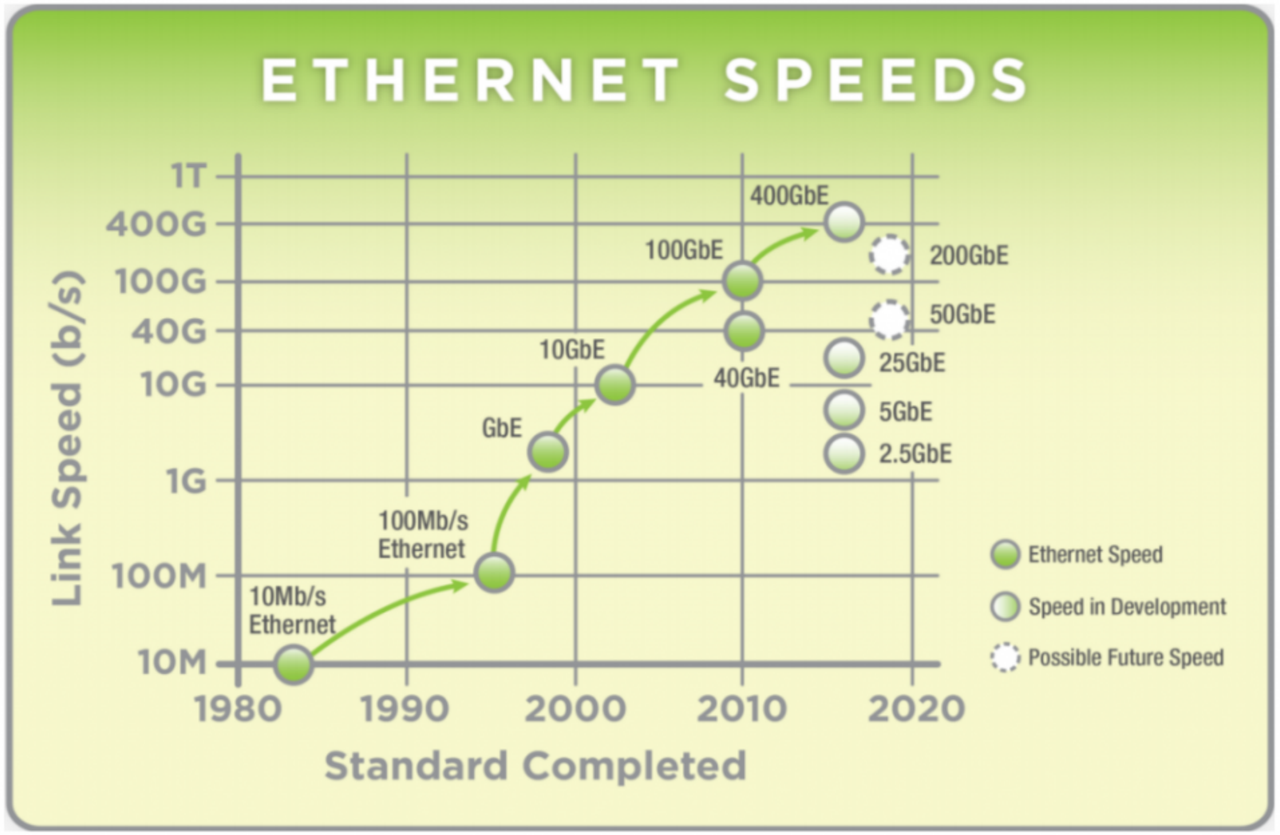
Hardware Flow Offload

What is it? Why you should matter?

Good News: Network Speed

The market is moving from 10 Gbit to 40/100 Gbit

- At 40 Gbit frame inter-arrival time is ~16 nsec
- At 100 Gbit frame inter-arrival time is ~6 nsec



X

Bad News: Nothing New Beside Speed

- In the past 15/20 years nothing really happened in networking hardware beside speed bump:
  - People talk about application protocols and metrics, NIC manufacturers about packet headers.
  - NICs are mostly stateless and the most they can do is to filter packets at header level.

Monitoring and Policy Enforcement

- Most network monitoring applications are based on the concept of connection (a.k.a. flow): a set of packets with the same 5-tuple.
- Firewalls, IPS/IDS, monitoring tools all work the same way:
  - Decode the packet header and hash it.
  - Find the hash bucket an update it.
  - Perform an action (e.g. compute a flow).

x

Flow Monitoring at 40/100G

- In the past years we demonstrated that it's possible to generate NetFlow in software at 100Gbit
    - Optimising the software to scale on multi-core processors
    - Leveraging on FPGA adapters for accelerating packet capture
- We created a new lightweight network probe named nProbe Cento

01/Dec/2015 16:00:25 [cento.cpp:513] Actual stats: 132'125'746 pps/0 drops

x

Flows and Hardware

- In order to accelerate flow processing advanced hardware NICs provide metadata that include <5 tuple, header hash>.
- Unfortunately the hash is often computed sub-optimally and thus it won't help much with collisions.
- Software application are responsible for doing all the rest (i.e. everything past packet decoding) that is still a lot.
- Question: can we offload some more tasks to hardware ? Could we leverage on hardware to keep flow state?

Hardware as a White Canvas

- Two years ago we have been asked by a network manufacturer company (Accolade Technology) to tell them what to implement in hardware to improve software applications, and they will do.
- ntop answer was to make firewalls,IDS/IPSs, monitoring faster by
  - Keeping flow state in hardware.
  - Periodically report flow information to software.
  - Execute actions based on flow state.
  - High capacity (no toy implementations).

What about DPI?

- ntop maintains an open source library named nDPI: it would be nice to make if faster.
- Deep Packet Inspection (DPI) requires to scan traffic data at line-rate
- Content scanning is CPU intensive and presents significant challenges to network analysis applications
- Overwhelmed by the traffic rates of 40/100G networks, it's more likely to loose traffic.

What about DPI in Hw?

- Offloading is not flexible
  - Hardware is not as flexible as software
  - Application protocols change every day
  - High development and manufacturing costs
- Hardware is designed for mostly static patterns, software is more dynamic and flexible.

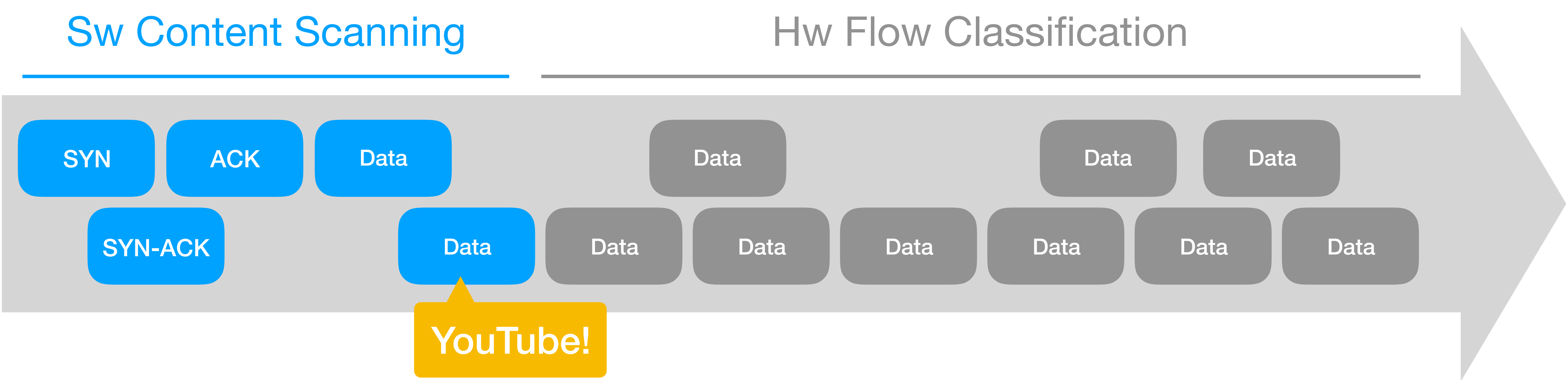
x

A Hybrid Solution

- FPGA adapters can be programmed to:
  - Keep flow state
  - Do basic flow classification and provide informations like hash, packets, bytes, first/last packet timestamp, tcp flags
- Software can be used for those activities that the FPGA cannot carry on (e.g. DPI and flow export)

Hardware-assisted DPI

- Software DPI on the initial flow packets until the L7 protocol is detected.
- Hardware can be instructed to drop/ bypass/prioritise flows based on the DPI decision. All in hardware.



X

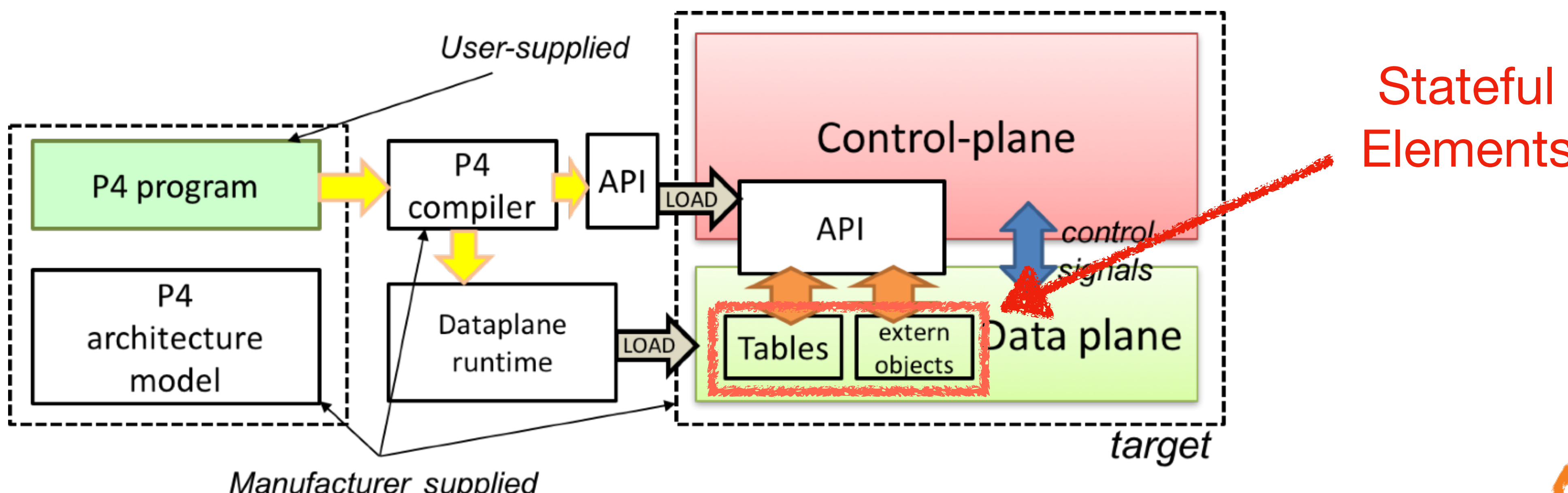
What About…

- NetFlow and flow-based monitoring
- Selective Packet-to-Disk: shunting.
- Flow-based filtering: DPI filtering or initial flow bytes (e.g HTTP header).

- Answer: we can accelerate them all using hardware flow state + actions.

Flow Offload Support [1/3]

- Available with PF_RING 7.0 currently supporting Accolade ANIC-Ku Series FPGA
- This work was triggered by Accolade but the PF_RING API is generic and we hope that other NIC manufacturers will implement something similar.
- The concept of "smartNIC" and p4.org include stateful elements that we would like to support in PF_RING when they will become available.



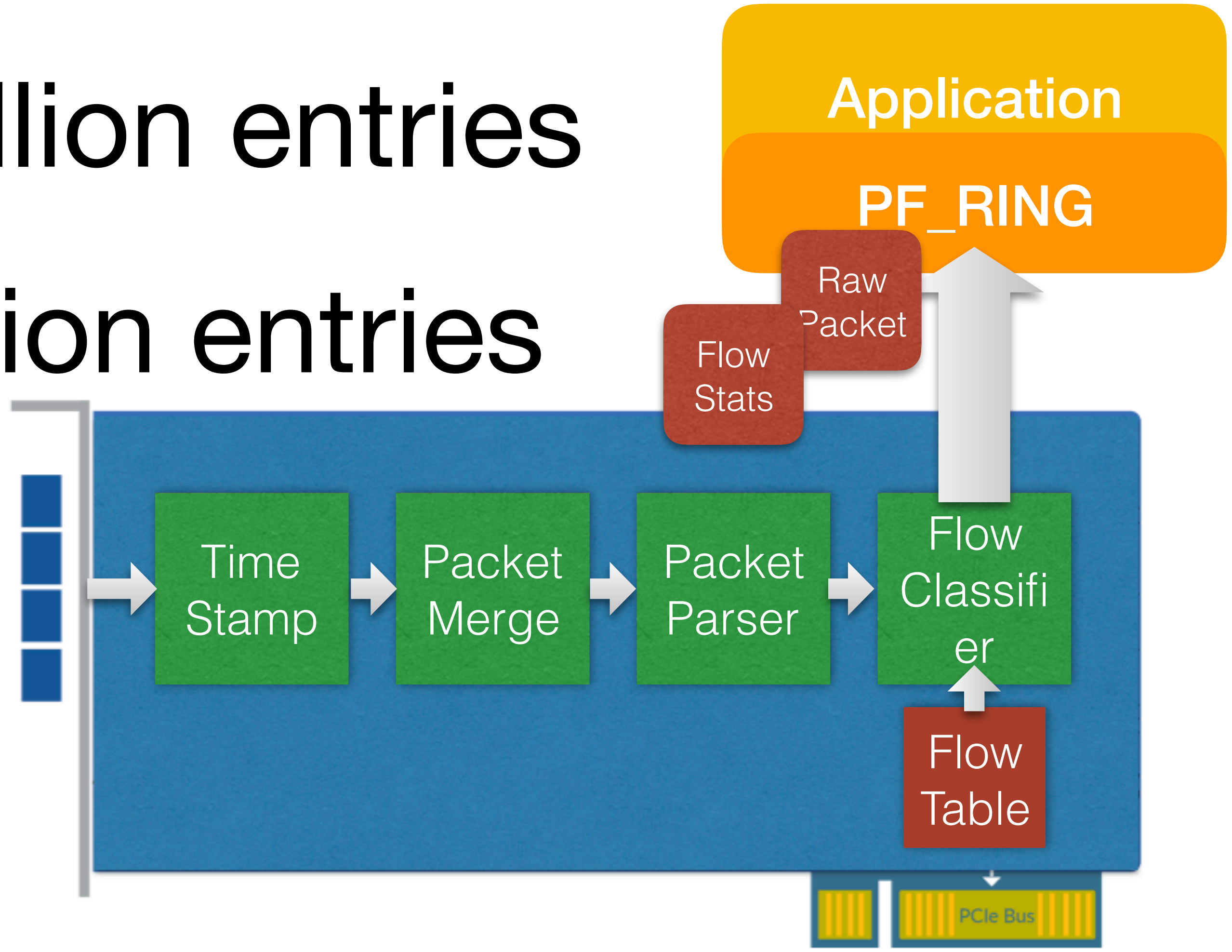
x

Flow Offload Support [2/3]

- Raw packets with unique Flow ID tag (no hashing with collisions as NICs usually do)
- Periodic flow messages (software configurable)
  - Flow Creation
  - Periodic Flow Updates (with counters)
  - Flow Deletion (inactivity)
- Ability to set (from software) the flow verdict on hardware so that future flows will stick to the decision made (drop/pass/shunt).

Flow Offload Support [3/3]

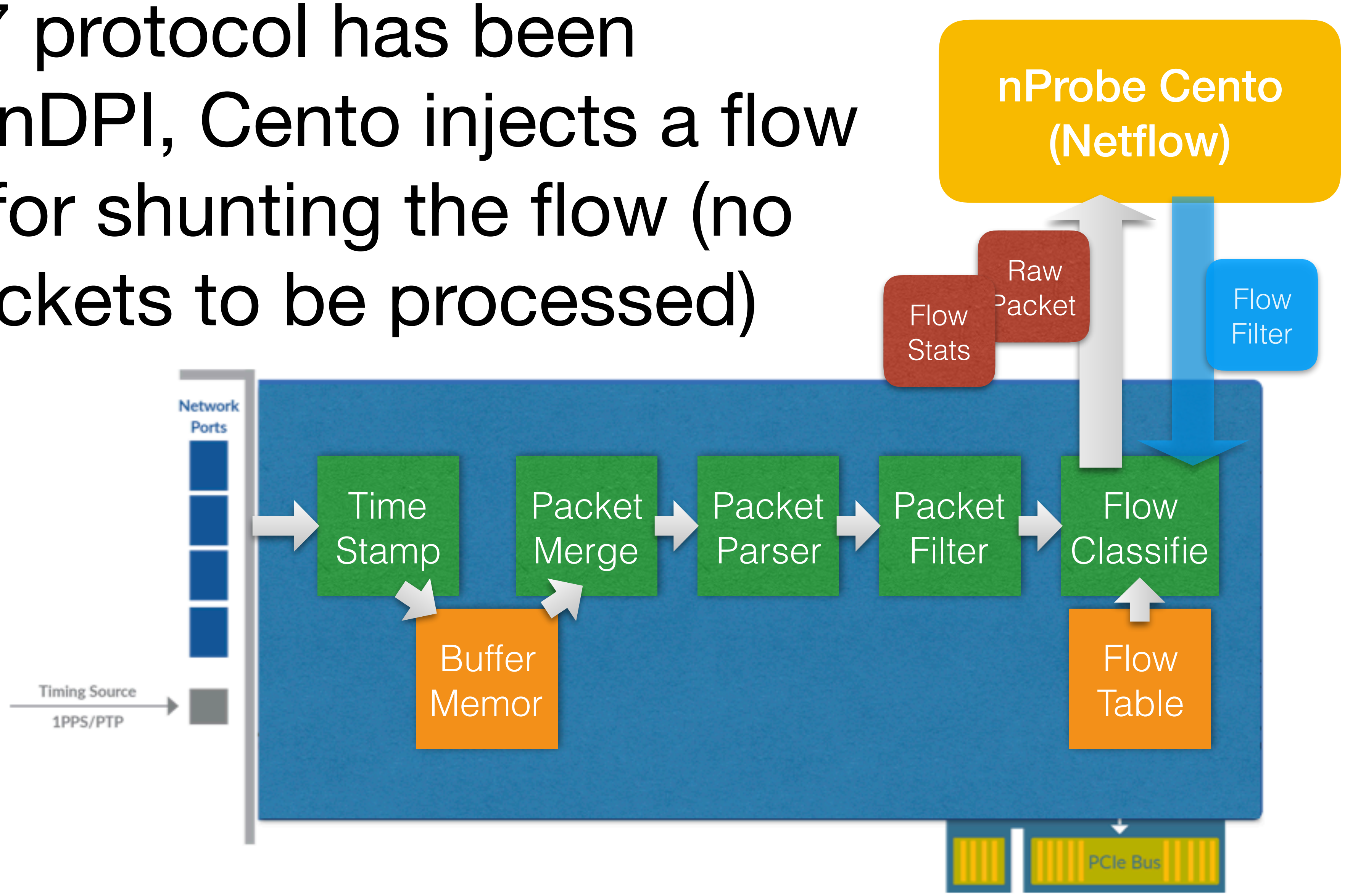
- Benchmarked at 4 x 10GE full bandwidth with 128B sized packets
- Adapters Flow Capacity:
  - 10/40 Gbit: 6 million entries
  - 100 Gbit: 32 million entries



X

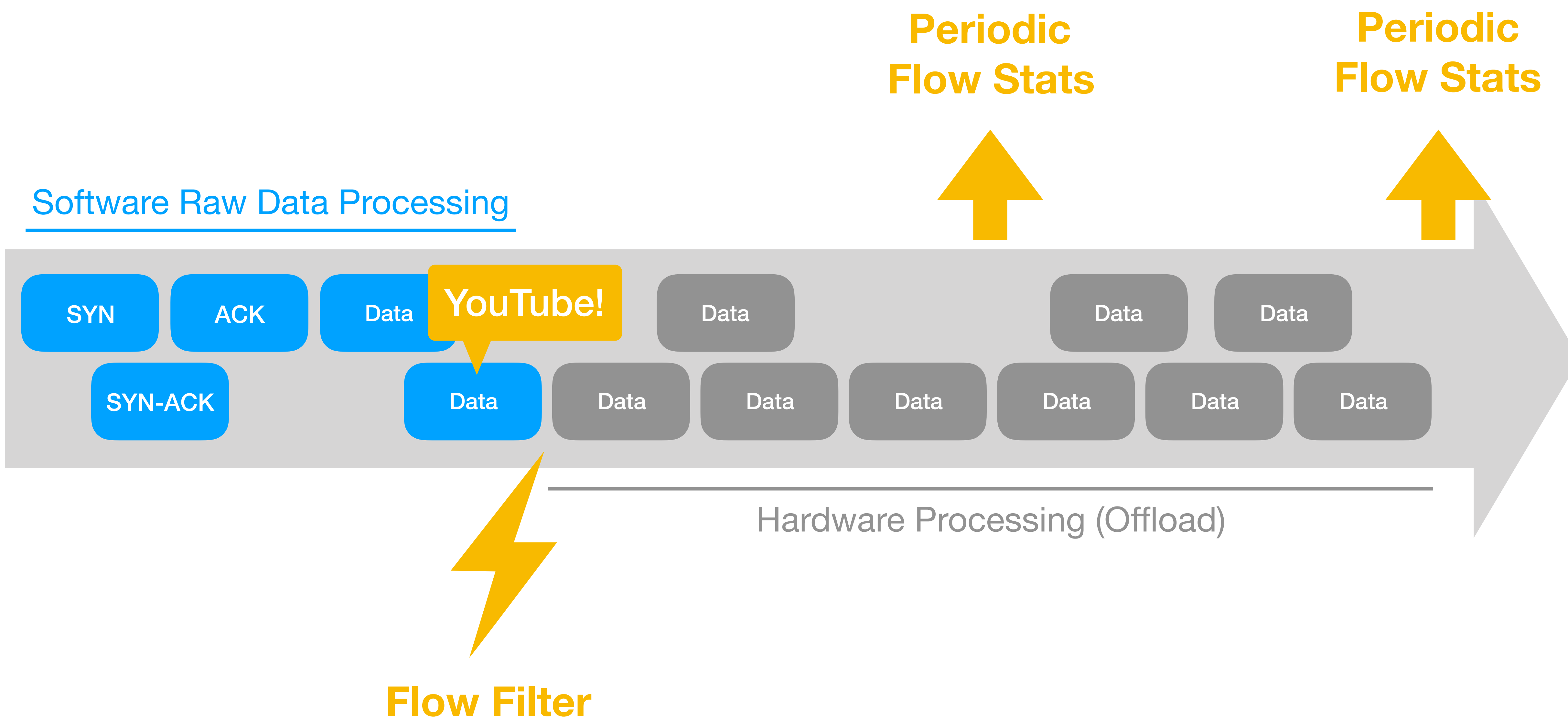
Cento with Flow Offload

- Cento receives both raw packets and periodic flow stats updates
- As soon a L7 protocol has been detected by nDPI, Cento injects a flow filtering rule for shunting the flow (no more raw packets to be processed)



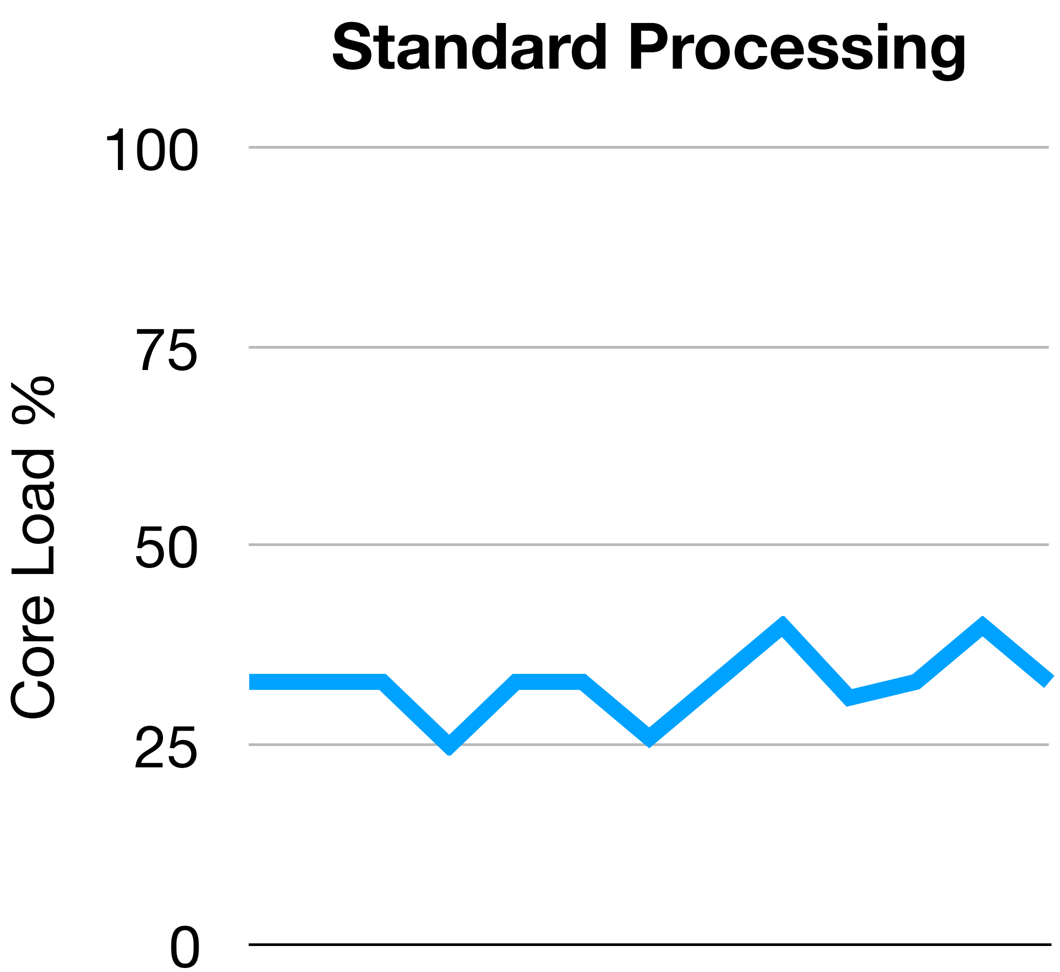
X

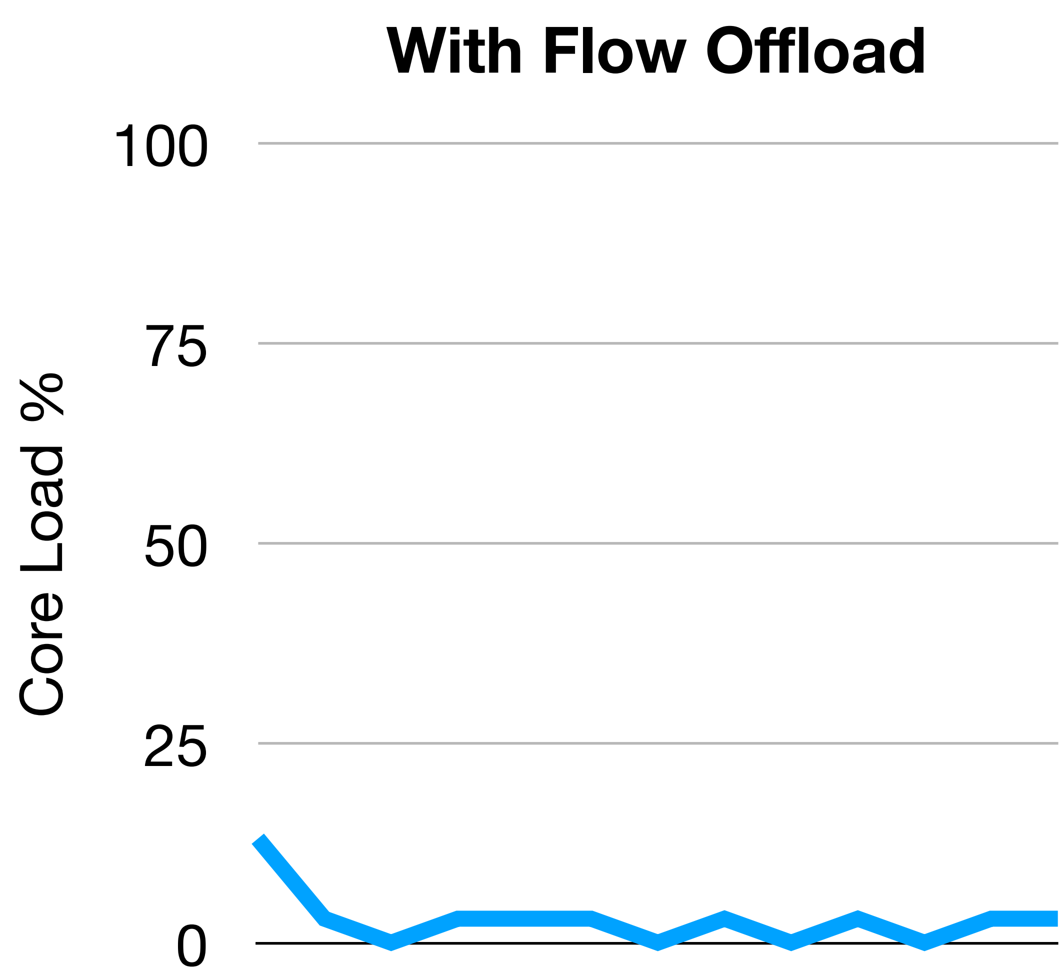
Hw-assisted DPI in Cento



Cento with Flow Offload - Performance [1/2]

- Test with real-life traffic, 5 Mpps, 500 flows on Intel E3 (single core)

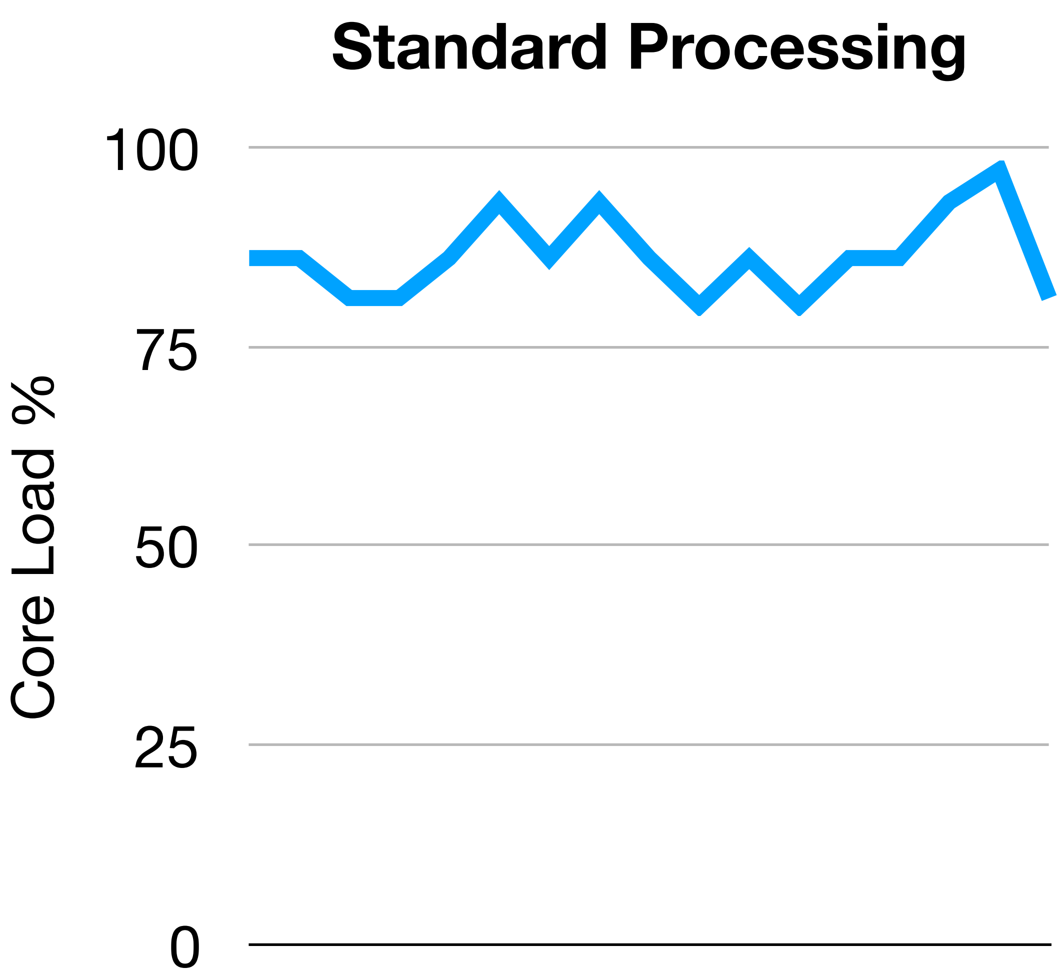


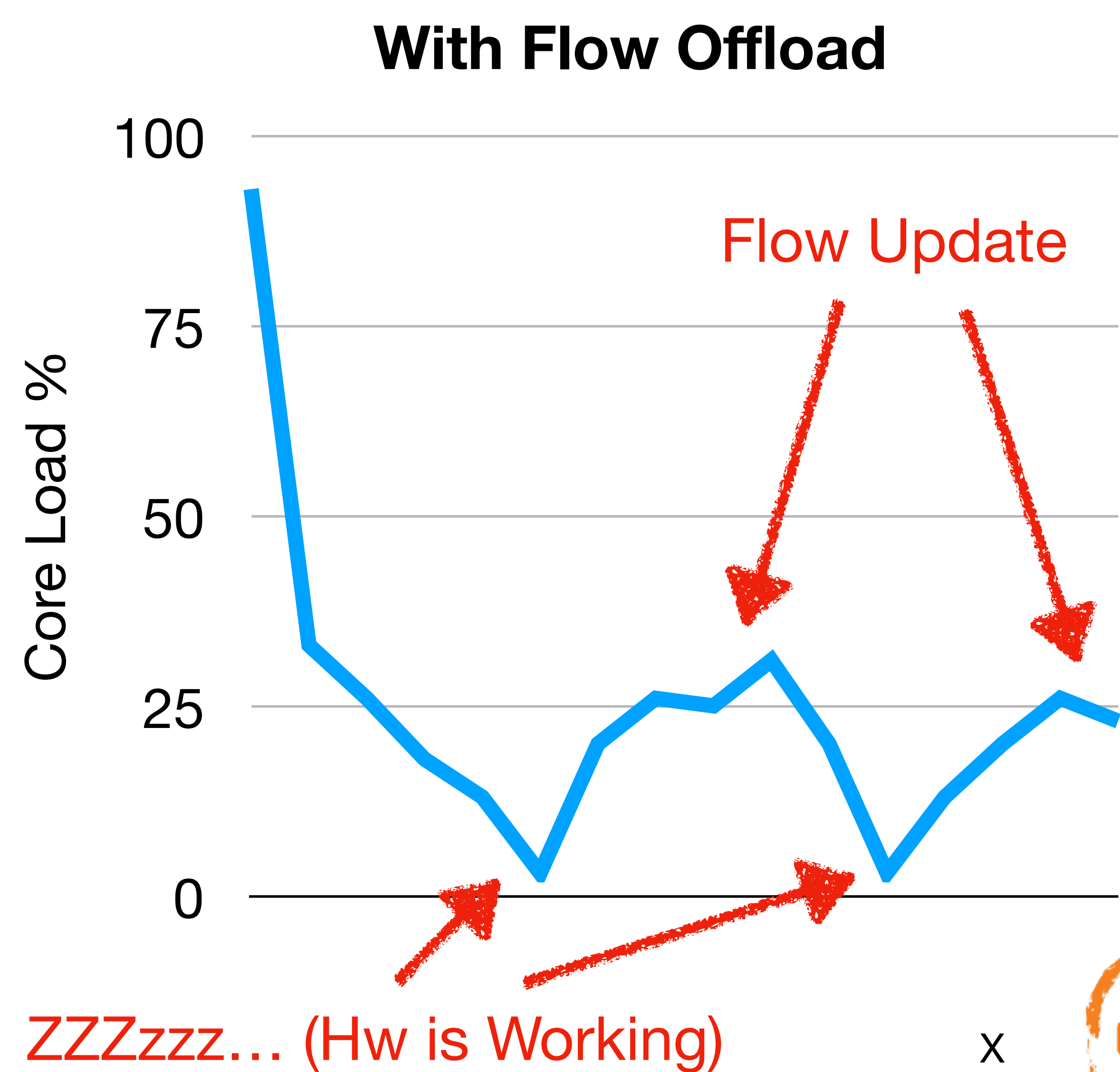


X

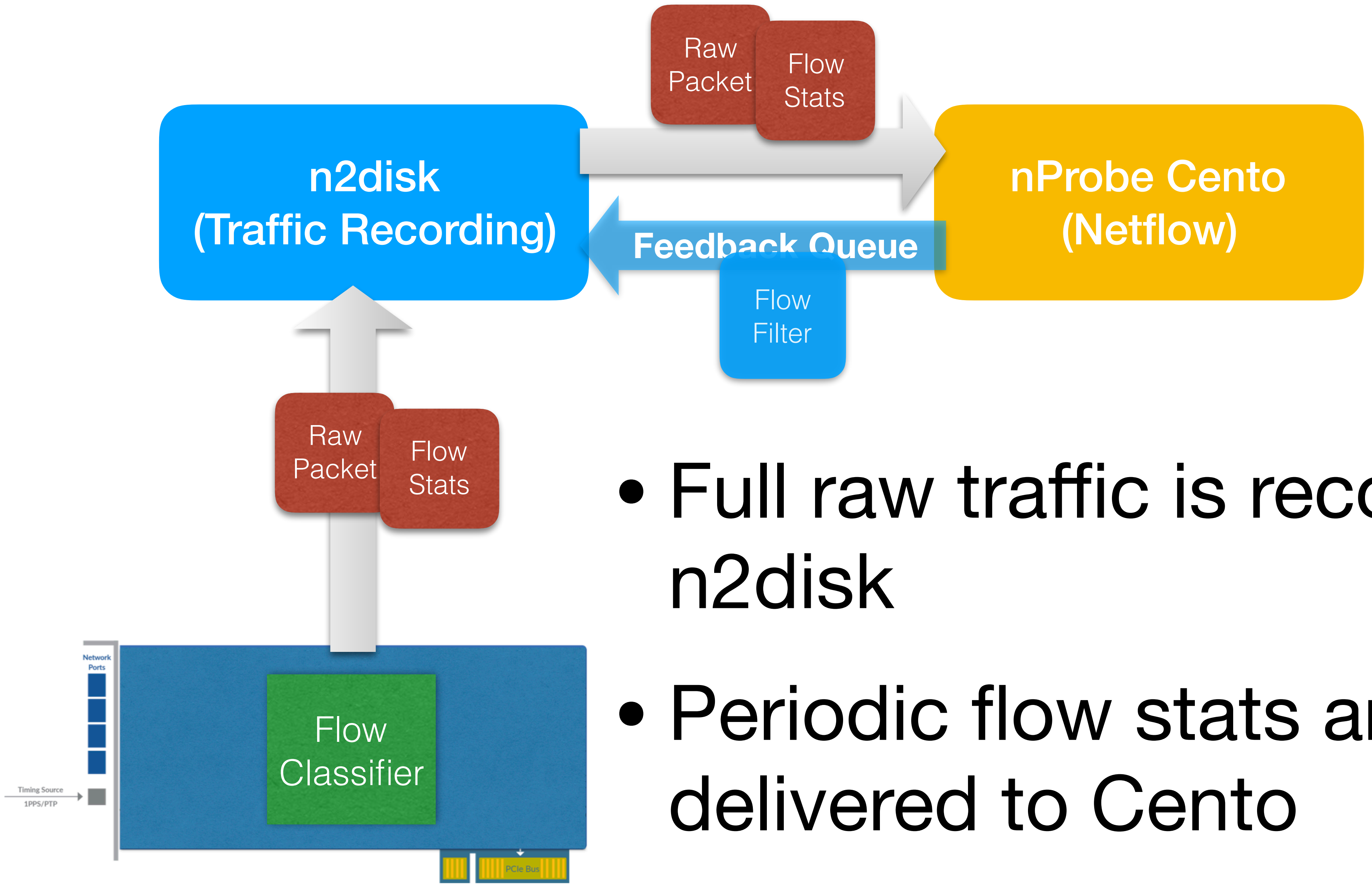
Cento with Flow Offload - Performance [2/2]

- Test with UDP traffic, 13 Mpps, 500K flows on Intel E3 (single core)



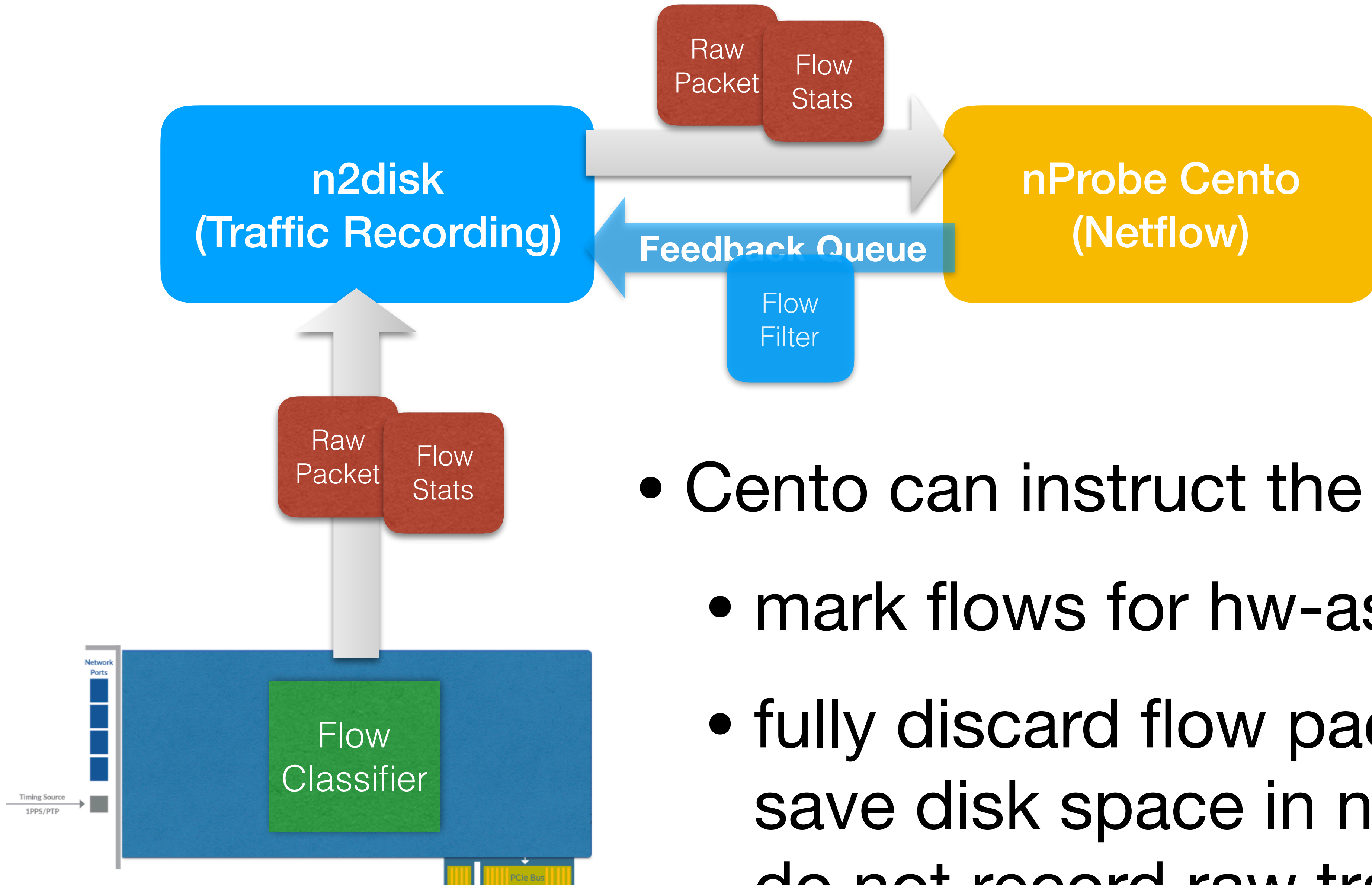


Netflow & Packet-to-Disk [1/2]



- Full raw traffic is recorded in n2disk
- Periodic flow stats are delivered to Cento
- Hw-assisted DPI with flow shunting by using markers

Netflow & Packet-to-Disk [2/2]

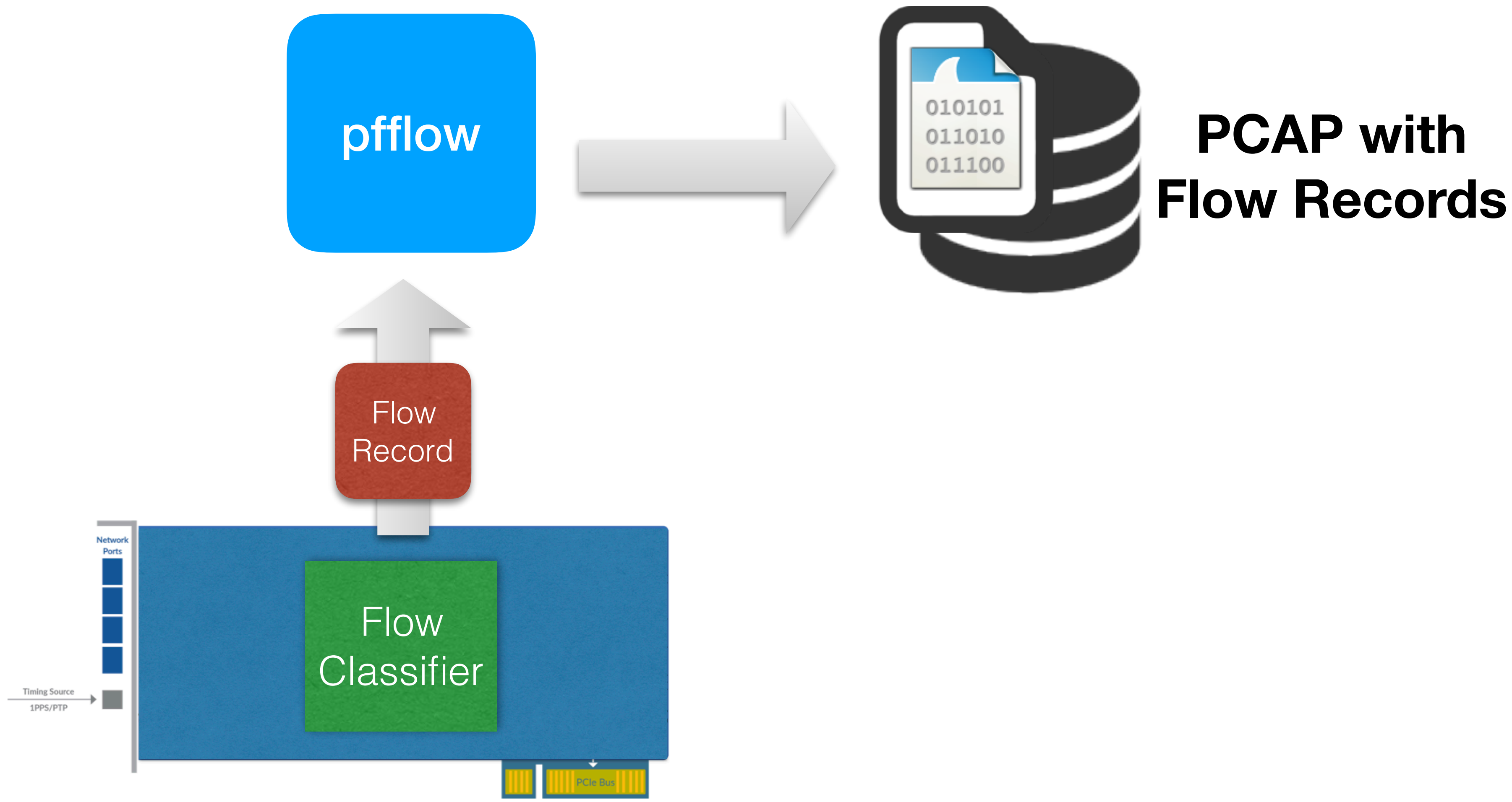


- Cento can instruct the card to:
  - mark flows for hw-assisted DPI
  - fully discard flow packets to save disk space in n2disk (e.g. do not record raw traffic for video streaming) leveraging on L7 protocol detection.

x

pfflow: Flow To Disk [1/3]

- pfflow: tiny application dumping Accolade flows to disk in PCAP format



X

pfflow: Flow To Disk [2/3]

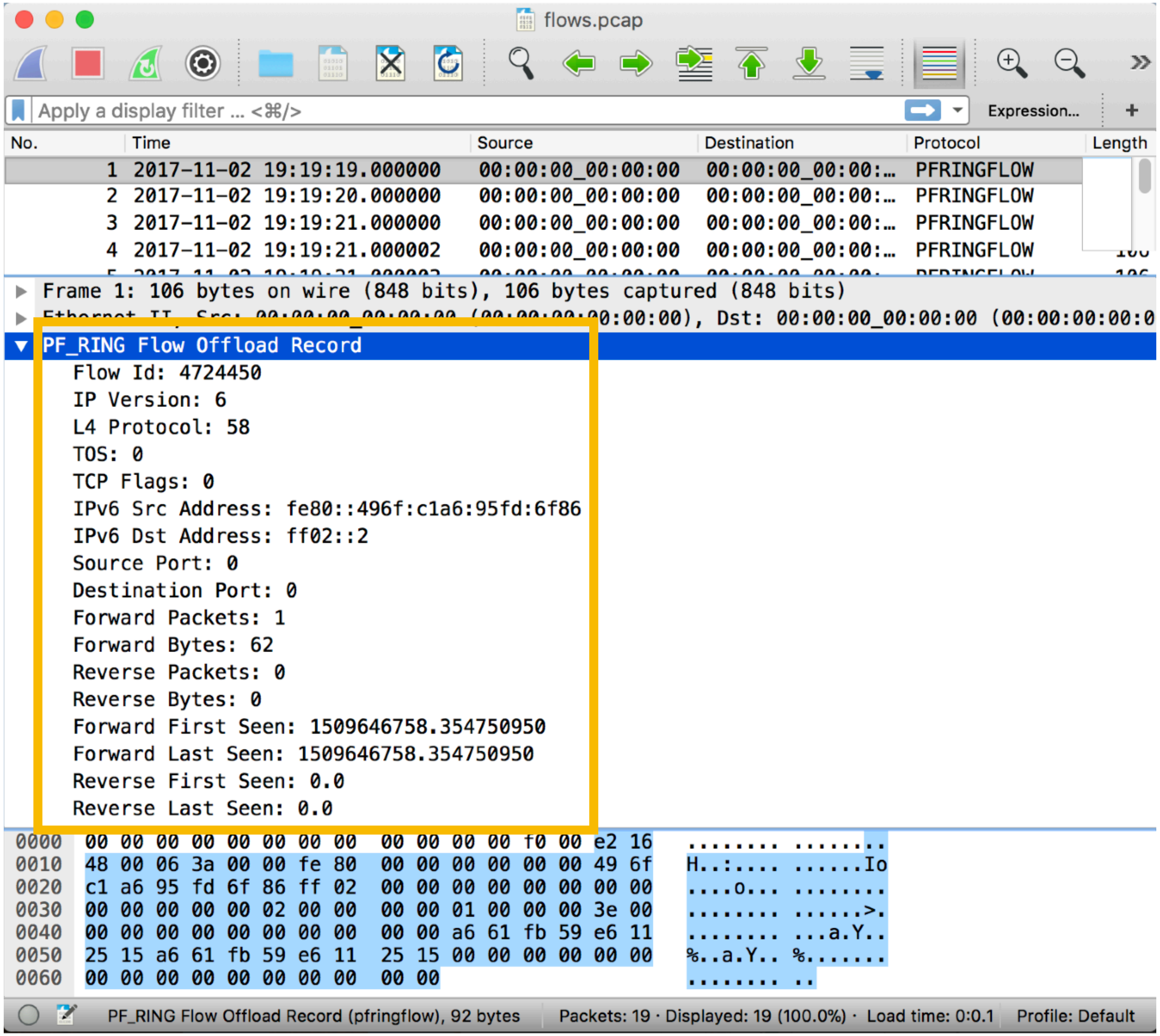
```
void processBuffer(const struct pfring_pkthdr *h,
                   const u_char *p, const u_char *user_bytes) {
  if (h->extended_hdr.flags & PKT_FLAGS_FLOW_OFFLOAD_UPDATE) {
    processFlow((generic_flow_update *) p);
    if (dumper != NULL)
       flowDump((struct pcap_pkthdr *) h,
                (generic_flow_update *) p);
  } else {
    processPacket(p, h->len, h->extended_hdr.pkt_hash);
  }
}

int main(int argc, char* argv[]) {
  pfring *pd = pfring_open("anic0", 1500 /* snaplen */,
                           PF_RING_FLOW_OFFLOAD);
  pfring_set_socket_mode(pd, recv_only_mode);
  pfring_loop(pd, processBuffer, (u_char*)NULL, wait_for_packet);
  pfring_close(pd);
}
```

pfflow: Flow To Disk [3/3]

```
void processFlow(generic_flow_update *flow) {
  char buf1[30], buf2[30];
  char *ip1, *ip2;

  if (flow->ip_version == 4){
    ip1 = _intoa(flow->src_ip.v4, buf1, sizeof(buf1));
    ip2 = _intoa(flow->dst_ip.v4, buf2, sizeof(buf2));
  } else {
    ip1 = (char *) inet_ntop(AF_INET6, &flow->src_ip.v6.s6_addr, buf1, sizeof(buf1));
    ip2 = (char *) inet_ntop(AF_INET6, &flow->dst_ip.v6.s6_addr, buf2, sizeof(buf2));
  }

  if (!quiet) {
    printf("Flow Update: flowID = %u "
           "srcIp = %s dstIp = %s srcPort = %u dstPort = %u protocol = %u tcpFlags = 0x%02X "
           "fwd: Packets = %u Bytes = %u FirstTime = %u.%u LastTime = %u.%u "
           "rev: Packets = %u Bytes = %u FirstTime = %u.%u LastTime = %u.%u\n",
           flow->flow_id, ip1, ip2, flow->src_port, flow->dst_port, flow->l4_protocol,
           flow->tcp_flags,
           flow->fwd_packets, flow->fwd_bytes, flow->fwd_ts_first.tv_sec,
           flow->fwd_ts_first.tv_nsec, flow->fwd_ts_last.tv_sec, flow->fwd_ts_last.tv_nsec,
           flow->rev_packets, flow->rev_bytes, flow->rev_ts_first.tv_sec,
           flow->rev_ts_first.tv_nsec, flow->rev_ts_last.tv_sec, flow->rev_ts_last.tv_nsec);
  }
}
```

X

PFRingFlow: Wireshark Flow Dissector [1/3]



X

PFRingFlow: Wireshark Flow Dissector [2/3]

```
-- create myproto protocol and its fields
p_pfringflow = Proto("PFRingFlow", "PF_RING Flow Offload Record")

local f_flow_id = ProtoField.uint32("pfringflow.flow_id", "Flow Id", base.DEC)
local f_ip_version = ProtoField.uint8("pfringflow.ip_version", "IP Version", base.DEC)
local f_l4_protocol = ProtoField.uint8("pfringflow.l4_protocol", "L4 Protocol", base.DEC)
local f_tos = ProtoField.uint8("pfringflow.tos", "TOS", base.DEC)
local f_tcp_flags = ProtoField.uint8("pfringflow.tcp_flags", "TCP Flags", base.DEC)
local f_src_ipv4 = ProtoField.ipv4("pfringflow.src_ipv4", "IPv4 Src Address")
local f_src_ipv6 = ProtoField.ipv6("pfringflow.src_ipv6", "IPv6 Src Address")
local f_dst_ipv4 = ProtoField.ipv4("pfringflow.dst_ipv4", "IPv4 Dst Address")
local f_dst_ipv6 = ProtoField.ipv6("pfringflow.dst_ipv6", "IPv6 Dst Address")
local f_src_port = ProtoField.uint16("pfringflow.src_port", "Source Port", base.DEC)
local f_dst_port = ProtoField.uint16("pfringflow.dst_port", "Destination Port", base.DEC)
local f_fwd_packets = ProtoField.uint32("pfringflow.fwd_packets", "Forward Packets", base.DEC)
local f_fwd_bytes = ProtoField.uint32("pfringflow.fwd_bytes", "Forward Bytes", base.DEC)
local f_rev_packets = ProtoField.uint32("pfringflow.rev_packets", "Reverse Packets", base.DEC)
local f_rev_bytes = ProtoField.uint32("pfringflow.rev_bytes", "Reverse Bytes", base.DEC)
  -- Timestamp format: (sec << 32) | (nsec)
local f_fwd_ts_first = ProtoField.string("pfringflow.fwd_ts_first", "Forward First Seen")
local f_fwd_ts_last = ProtoField.string("pfringflow.fwd_ts_last", "Forward Last Seen")
local f_rev_ts_first = ProtoField.string("pfringflow.rev_ts_first", "Reverse First Seen")
local f_rev_ts_last = ProtoField.string("pfringflow.rev_ts_last", "Reverse Last Seen")


p_pfringflow.fields = { f_flow_id, f_ip_version, f_l4_protocol, f_tos, f_tcp_flags,
                      f_src_ipv4, f_src_ipv6, f_dst_ipv4, f_dst_ipv6,
                      f_src_port, f_dst_port, f_fwd_packets, f_fwd_bytes, f_rev_packets, f_rev_bytes,
                      f_fwd_ts_first, f_fwd_ts_last, f_rev_ts_first, f_rev_ts_last
}
```

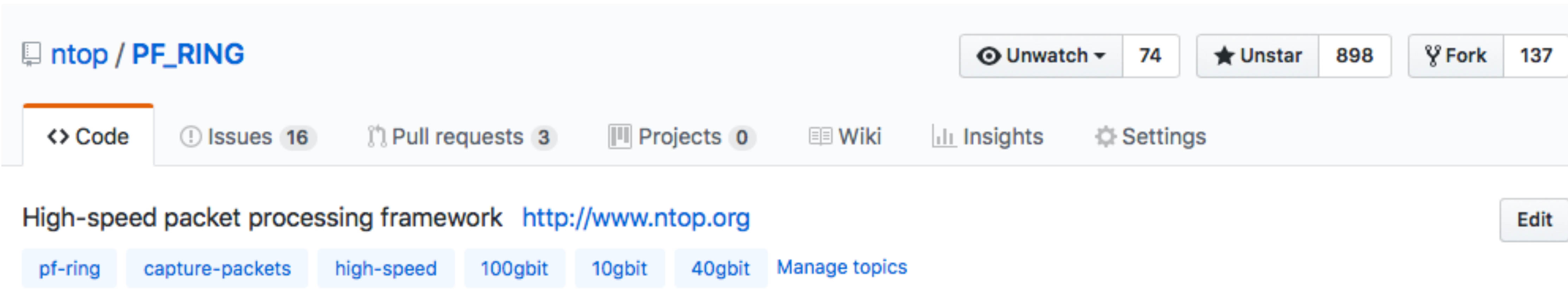
X

PFRingFlow: Wireshark Flow Dissector [3/3]

```
function p_pfringflow.dissector (buf, pkt, root)
   local sec, nsec, sec_offset

  if buf:len() == 0 then return end
   pkt.cols.protocol = p_pfringflow.name

   -- create subtree for pfringflow
   subtree = root:add(p_pfringflow, buf(0))
   offset = 0

   -- add protocol fields to subtree
   subtree:add_le(f_flow_id, buf(offset, 4))
   offset = offset + 4

  ……
   sec_offset = offset
   sec = buf(offset, 4):le_uint()
   offset = offset + 4

   nsec = buf(offset, 4):le_uint()
   offset = offset + 4

   subtree:add(f_rev_ts_last, buf(sec_offset, 8), sec.."."..nsec)

end

-- Initialization routine
function p_pfringflow.init()
end

-- 0x0F00 = 61440
local eth_dissector_table = DissectorTable.get("ethertype")
dissector = eth_dissector_table:get_dissector(61440)


eth_dissector_table:add(61440, p_pfringflow)
```

X

Code Availability



- https://github.com/ntop/PF_RING
  - https://github.com/ntop/PF_RING/blob/dev/userland/examples/pfflow.c
  - https://github.com/ntop/PF_RING/tree/dev/userland/wireshark/plugins