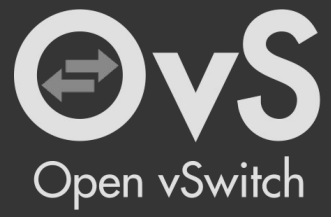


Running Open vSwitch on a large CPU system

OVS/OVN Conference 2024

Eelco Chaudron
Senior Principal Software Engineer

Red Hat

Agenda

- Objective;
  - Evaluate how high CPU count impacts revalidation and ovs-vswitchd functionality under normal operating conditions.
- Focus Areas;
  - Revalidation process
  - Interface statistics impact on run time
  - Simulated POD creation
  - getifaddrs() run time

- Hardware:
  - Dell PowerEdge R6515
  - AMD EPYC 7702P 64-Core/128-Threads Processor
  - Memory 16GB DDR4
  - Storage 446.63 GB SATA/SSD
- Software:
  - RHEL 9.5
  - Open vSwitch 3.3
- Number of CPUs:
  - 128 CPUs; all threads are enabled
  - 64 CPUs; the number of cores per CCD has been reduced to FOUR(2+2) in the BIOS.

The Revalidation Process

Red Hat

- Revalidation is managed by the revalidator threads
  - Making sure the installed datapath flows match the configured OpenFlow rules
  - Updating OpenFlow packet and byte statistics
  - Removing idle datapath flows

- More details in the following blog:
  Open vSwitch: The revalidator process explained

- By default, if *n-revalidator-threads* is not configured, for a 128 CPU system, 33 revalidator threads are started.
  - The number of CPUs available to ovs-vswitchd divided by four, plus one.

- We will compare data with 4, 32 and 128 revalidator threads.

- Simple Open vSwitch configuration:
  - Two Network namespace simulating endpoints
  - veth interface to connect endpoints to OVS
  - Scapy for generating the traffic
  - Maximum datapath flows to exists (200k)

- Using taskset to bind the Scapy script to a specific CPU (set).

```
# ip netns exec ns_h1 bash
# taskset -c 1 python traffic.py
```

NETNS configuration details

```
ip link add h1_ovs type veth peer name h1_eth
ip link add h2_ovs type veth peer name h2_eth

ip link set dev h1_ovs up
ip link set dev h2_ovs up

ip netns add ns_h1
ip netns add ns_h2

ip link set h1_eth netns ns_h1
ip link set h2_eth netns ns_h2

ip -n ns_h1 link set dev lo up
ip -n ns_h1 link set dev h1_eth up
ip -n ns_h1 address add 192.168.9.1/24 dev h1_eth

ip -n ns_h2 link set dev lo up
ip -n ns_h2 link set dev h2_eth up
ip -n ns_h2 address add 192.168.9.2/24 dev h2_eth
```

OVS configuration details

```
ovs-vsctl add-br ovs_pvp_br0
ovs-vsctl add-port ovs_pvp_br0 h1_ovs
ovs-vsctl add-port ovs_pvp_br0 h2_ovs
ovs-appctl upcall/set-flow-limit 200000

ovs-ofctl del-flows ovs_pvp_br0
ovs-ofctl add-flow ovs_pvp_br0 in_port=h1_ovs,arp,action=h2_ovs
ovs-ofctl add-flow ovs_pvp_br0 in_port=h1_ovs,ip,nw_dst=192.168.9.2,action=h2_ovs
ovs-ofctl add-flow ovs_pvp_br0 in_port=h2_ovs,arp,action=h1_ovs
ovs-ofctl add-flow ovs_pvp_br0 in_port=h2_ovs,ip,nw_dst=192.168.9.1,action=h1_ovs

ip netns exec ns_h1 ping 192.168.9.2

python -c 'for i in range(0, 200000): \
  print("add in_port=h1_ovs,ip,nw_dst=10.{}.{}.{}/32,action=h2_ovs".
    format((i >>16) & 0xff, (i >> 8) & 0xff, i & 0xff))' | \
  ovs-ofctl add-flow ovs_pvp_br0 -
```

Scapy script

```
from scapy.all import *

  def ip_from_int(decimal):
      ip_int = 0x0a000000 + decimal;
      return "{}.{}.{}.{}".format(ip_int >> 24 & 0xff,
                                  ip_int >> 16 & 0xff,
                                  ip_int >> 8 & 0xff,
                                  ip_int & 0xff);

  pkt_list = []
  eth = Ether(dst="36:c2:88:9e:19:b1", src="00:00:00:00:00:01")
  ip = IP(dst="10.0.0.1", src="10.200.0.1")
  udp = UDP(dport=69)

  print("Building packet list...")
  for i in range(0, 100000):
      ip.dst=ip_from_int(i)
      pkt_list.append(eth / ip / udp)
      if (i % 1000) == 0 and i != 0:
          print("Frame: {}".format(i))

  print("Sending packets...")

  sendpfast(pkt_list, iface="h1_eth", loop=0)
```

- The following tests where ran with 4, 32, and 128 revalidator threads:
  - 200k datapath flows, with traffic generated from a single CPU.
  - 200k datapath flows + 10x a second a flow revalidation sweep.
  - 200k datapath flows, with traffic generated by all CPU cores.

- Existing USDT probes and the reval_monitor.py script is used for the initial analysis.

200k datapath flows with traffic arriving on a single CPU

- Results with 4 revalidator threads:

```
=> dump_duration:
# NumSamples = 641; Min = 158.00; Max = 185.00
# Mean = 169.695788; Variance = 26.277190; SD = 5.126128; Median 170.000000
# each ▌ represents a count of 1
  158.0000 -   160.7000 [    26]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  160.7000 -   163.4000 [    67]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  163.4000 -   166.1000 [    72]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  166.1000 -   168.8000 [    95]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  168.8000 -   171.5000 [   143]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  171.5000 -   174.2000 [   123]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  174.2000 -   176.9000 [    48]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  176.9000 -   179.6000 [    54]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  179.6000 -   182.3000 [    10]: ▌▌▌▌▌▌▌▌▌▌
  182.3000 -   185.0000 [     3]: ▌▌▌
```

❏ Results with 32 revalidator threads:

```
=> dump_duration:
# NumSamples = 626; Min = 171.00; Max = 196.00
# Mean = 180.718850; Variance = 25.441721; SD = 5.043979; Median 180.000000
# each ▌ represents a count of 1
  171.0000 -   173.5000 [    44]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  173.5000 -   176.0000 [   105]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  176.0000 -   178.5000 [    93]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  178.5000 -   181.0000 [   116]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  181.0000 -   183.5000 [    70]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  183.5000 -   186.0000 [   110]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  186.0000 -   188.5000 [    51]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  188.5000 -   191.0000 [    25]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  191.0000 -   193.5000 [     9]: ▌▌▌▌▌▌▌▌▌
  193.5000 -   196.0000 [     3]: ▌▌▌
```

❑ Results with 128 revalidator threads:

```
=> dump_duration:
# NumSamples = 628; Min = 169.00; Max = 195.00
# Mean = 183.399682; Variance = 16.771783; SD = 4.095337; Median 184.000000
# each ▌ represents a count of 2
  169.0000 -   171.6000 [     3]: ▌
  171.6000 -   174.2000 [    14]: ▌▌▌▌▌▌▌
  174.2000 -   176.8000 [    29]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  176.8000 -   179.4000 [    63]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  179.4000 -   182.0000 [   118]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  182.0000 -   184.6000 [   119]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  184.6000 -   187.2000 [   193]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  187.2000 -   189.8000 [    65]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
  189.8000 -   192.4000 [    22]: ▌▌▌▌▌▌▌▌▌▌▌
  192.4000 -   195.0000 [     2]: ▌
```

- So more threads, show slight increase in run time.
  - 169ms, 181ms, 183ms.
- Further analysis was done using the kernel_delay.py and analyze_perf_pmd_syscall.py scripts.
- Conclusion; it's due to the a lock contention:

```
#0  ovs_mutex_lock_at() at ./ovs/lib/ovs-thread.c:76
#1  nl_dump_next() at ./ovs/lib/netlink-socket.c:1257
#2  dpif_netlink_flow_dump_next() at ./ovs/lib/dpif-netlink.c:1957
#3  dpif_flow_dump_next() at ./ovs/lib/dpif.c:1145
```

- Further research should be done to see if we can optimize nl_dump_next() code:

```
bool
nl_dump_next(struct nl_dump *dump, struct ofpbuf *reply, struct ofpbuf *buffer)
{
    ...
    ...

    if (!buffer->size) {
        ovs_mutex_lock(&dump->mutex);
        if (!dump->status) {
            /* Take the mutex here to avoid an in-kernel race.  If two threads
             * try to read from a Netlink dump socket at once, then the socket
             * error can be set to EINVAL, which will be encountered on the
             * next recv on that socket, which could be anywhere due to the way
             * that we pool Netlink sockets.  Serializing the recv calls avoids
             * the issue. */
            dump->status = nl_dump_refill(dump, buffer);
        }
        retval = dump->status;
        ovs_mutex_unlock(&dump->mutex);
    }
    ...
    ...
}
```

200k datapath flows + revalidation sweep

- Runtime with 4, 32, and 128 revalidator threads:
  - 334ms, 209ms, 224ms
- More revalidation threads will speedup the actual revalidation process.
- However, to many will hit the same nl_dump_next() lock contention.

200k datapath flows with traffic arriving on multiple CPUs

- Why this test?
  - When experimenting we found that the OOM killer kicked in.
  - Memory was depleted due to massive *`sw_flow_stat`* allocations.

- What are these *`sw_flow_stat`* structures?
  - They hold per-CPU datapath flow statistics
  - 32-byte in size
  - Allocated on demand, when traffic is processed on the CPU.

- How much memory they consume for 200k datapath flows?
  - On a 128-CPU system: 782 MB
  - On a 256-CPU system: 1.5 GB

- Is the amount of memory the (main) problem?
  - No

- The main problem;
  - Time spent accumulating data due to non-linear memory storage.
  - Increases latency in processing data, i.e. getting flow statistics from the kernel to userspace.

- The increased latency when adding more *`sw_flow_stats`* structures leads to gradual increase in the *`dump_duration`*.

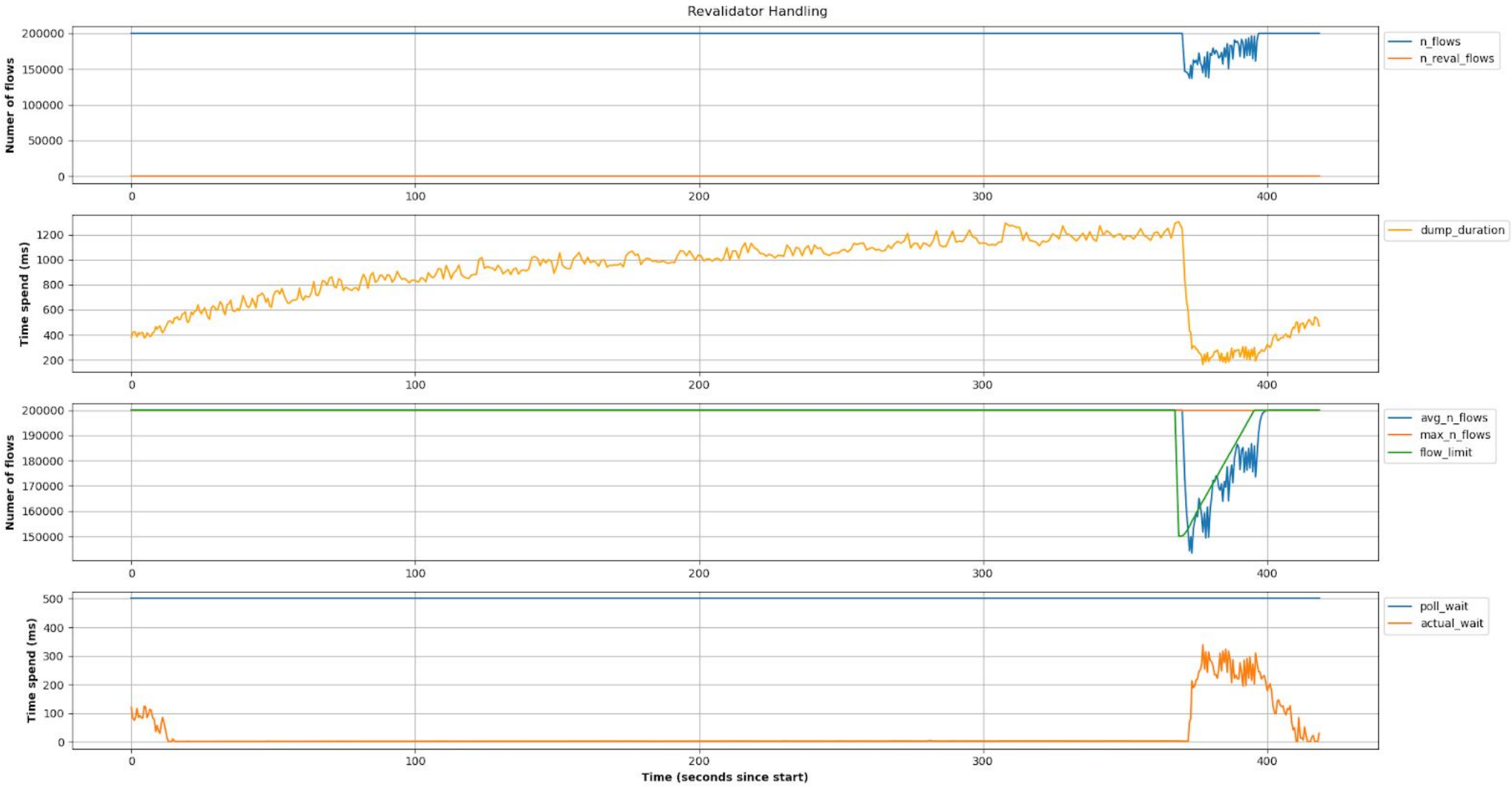
- This continues till we hit a threshold (1.3 second) at which the maximum datapath flows are reduced.

- Note that this is nothing new, it also exists in a 64-CPU system.

- The next slides will show some graphs while running these tests
  - Interesting observation; it takes longer to reach the threshold with a lower number of revalidator threads. I did not investigate why.

200k datapath flows / Rx on multiple CPUs; 4 revalidator threads

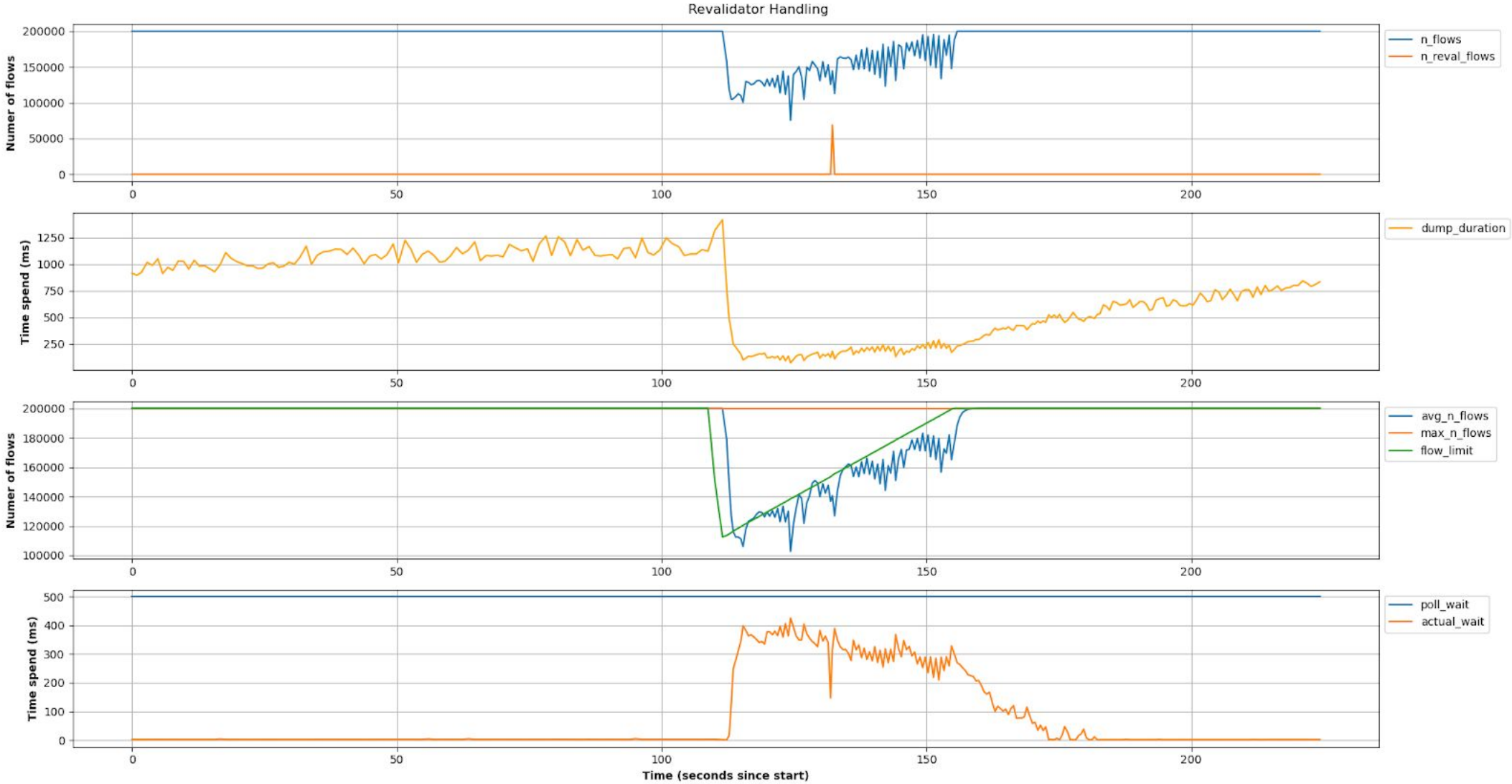
- 4 revalidator threads:



Red Hat

200k datapath flows / Rx on multiple CPUs; 32 revalidator threads

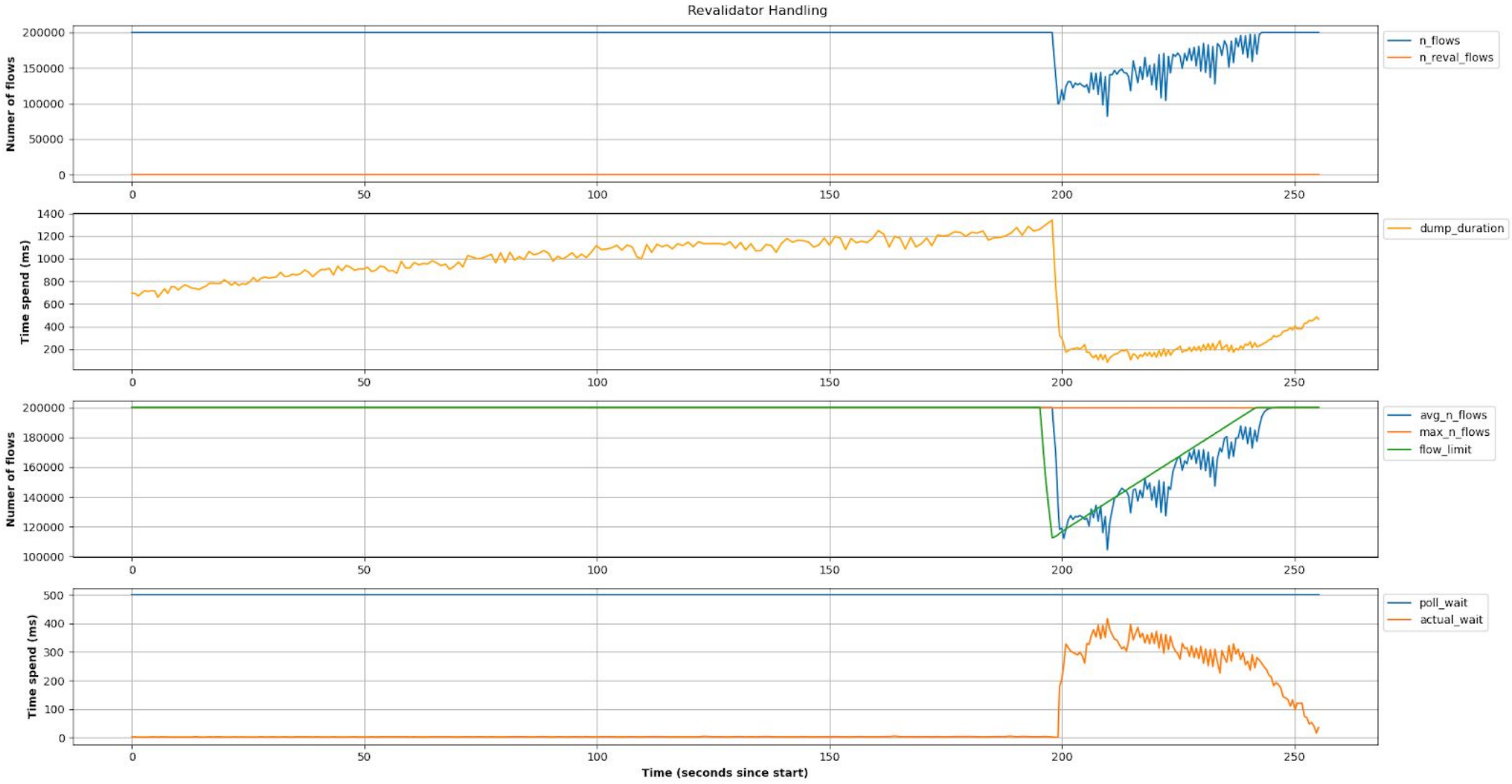
- 32 revalidator threads:



Red Hat

200k datapath flows / Rx on multiple CPUs; 128 revalidator threads

- 128 revalidator threads:



Red Hat

Kernel Interface Statistics Impact

- ovs-vswitchd periodically gathers all interface statistics (± every 5 seconds).
- Different kernel net drivers utilize different per-CPU statistic counters
  - We use veth drivers, which have specific per-CPU counters.
- Per-CPU netdev statistics are linear in memory.
- The Linux kernel also maintains certain exception counters in the *net_device_core_stats*.

```
/* per-cpu stats, allocated on demand.
 * Try to fit them in a single cache line, for dev_get_stats() sake.
 */
struct net_device_core_stats {
      unsigned long    rx_dropped;               IP STATS: RX: dropped
      unsigned long    tx_dropped;               IP STATS: TX: dropped
      unsigned long    rx_nohandler;             IP STATS: nohandler
      unsigned long    rx_otherhost_dropped;     IP STATS: otherhost
} __aligned(4 * sizeof(unsigned long));
```

For measuring the impact:

- Configure 1024 veth pairs; one side in its own netns, the other side in OVS.
- Add two USDT probes to ovs-vswitchd (see next slide) measuring the time it takes to update the interface statistics.
- Use a modified version of the bride_loop.bt script.
- Run for five minutes on a 64-core and 128-core configuration.
- One run without the exception stats being present, one with them being present.

Additional USDT probes:

```
--- a/vswitchd/bridge.c
+++ b/vswitchd/bridge.c
@@ -48,6 +48,7 @@
 #include "openvswitch/meta-flow.h"
 #include "openvswitch/ofp-print.h"
 #include "openvswitch/ofpbuf.h"
+#include "openvswitch/usdt-probes.h"
 #include "openvswitch/vconn.h"
 #include "openvswitch/vlog.h"
 #include "ovs-lldp.h"
@@ -3131,6 +3132,7 @@ run_stats_update(void)
             struct bridge *br;

             stats_txn = ovsdb_idl_txn_create(idl);
+            OVS_USDT_PROBE(run_stat_update, br_loop_start);
             HMAP_FOR_EACH (br, node, &all_bridges) {
                 struct port *port;
                 struct mirror *m;
@@ -3148,6 +3150,7 @@ run_stats_update(void)
                     mirror_refresh_stats(m);
                 }
             }
+            OVS_USDT_PROBE(run_stat_update, br_loop_done);
             refresh_controller_status();
         }
```

- 64-CPU result:

```
Average time: 0.063006219 seconds

@statistics_loop_time in micro seconds:
[60000, 70000)        62 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
```

- 128-CPU result:

```
Average time: 0.104204051 seconds

@statistics_loop_time in micro seconds:
[90000, 100000)       41 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[100000, 110000)       0 |                                                    |
[110000, 120000)      20 |@@@@@@@@@@@@@@@@@@@@@@@@@                           |
```

- Doubling the number of CPU result in slightly less than doubling the run time (40%).

- This is outside the control of OVS as these per-CPU counters are a driver specific implementation.

Red Hat

- Comparing with and without exception counters on a 128-CPU system

```
Average time: 0.104204051 seconds (without)

@statistics_loop_time in micro seconds:
[90000, 100000)       41 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[100000, 110000)       0 |                                                    |
[110000, 120000)      20 |@@@@@@@@@@@@@@@@@@@@@@@@@                           |

Average time: 0.104665879 seconds (with exception counters)

@poll_block_wait_time:
[90000, 100000)       26 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@          |
[100000, 110000)      32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[110000, 120000)       0 |                                                    |
[120000, 130000)       0 |                                                    |
[130000, 140000)       1 |@                                                   |
[140000, 150000)       0 |                                                    |
[150000, 160000)       5 |@@@@@@@@                                            |
```

- There's a difference of 461,828ns on average ≈ 451ns per interface.

- On a 64-Core system this was even slower, however I did not further research why. 5,089,948ns (5ms!) on average ≈ 4970ns per interface.

Simulated POD Creation

- The objective is to assess whether the creation and teardown of PODs have any effect on the main thread.
- Our SUT does not have enough memory for a OCP cluster so we use standalone OVN to simulate this.
- The test is defined as follows:
  - We create 128 OVN switch ports simulating 128 PODs.
  - We continuously add and delete 10 additional PODs.
  - The cycle takes about 1.3 seconds (typical large OCP cluster creates around 6 PODs per seconds).
  - This test does not include any actual traffic.
- For the initial analysis we use kernel_delay.py on ovs-vswitchd's main thread.

- 64-Cores (condensed output):

```
 4472 ovs-vswitchd     [SYSCALL STATISTICS]
NAME                NUMBER       COUNT          TOTAL ns           MAX ns
sendmsg                 46       99109     4,958,977,988        89,797,200
recvmsg                 47      115014     1,666,039,446        47,172,400
sendto                  44         915       863,941,291        42,862,807
...
TOTAL( - poll):                   412283     8,295,257,784
```

- 128-Cores (condensed output):

```
 3765 ovs-vswitchd     [SYSCALL STATISTICS]
NAME                NUMBER       COUNT          TOTAL ns           MAX ns
sendmsg                 46       91052     4,461,157,621        58,967,791
recvmsg                 47      107575     3,307,665,973        57,098,125
sendto                  44         932       485,015,325         8,727,121
...
TOTAL( - poll):                   377964     8,721,298,017
```

- So the 128-Cores instance spends more time on ***recvmesg()***
  - ≈ 50%

Simulated POD creation; Additional Analysis

- We used analyze_perf_pmd_syscall.py script to the call trace:

```
#0  netdev_get_addrs() at ./ovs/lib/netdev.c:2314
#1  netdev_linux_get_addr_list() at ./ovs/lib/netdev-linux.c:3540
#2  netdev_get_addr_list() @ netdev_get_addr_list at ./ovs/lib/netdev.c:1518
#3  verify_prefsrc() at ./ovs/lib/ovs-router.c:182
#4  ovs_router_insert__() at ./ovs/lib/ovs-router.c:292
```

- 64-Cores (condensed output):

```
=> Syscall: recvmsg/47 (duration in microseconds):
# NumSamples = 8386; Min = 5.15; Max = 41923.33
# Mean = 85.122323; Variance = 900372.542002; SD = 948.879625; Median 61.255000
# each ▌ represents a count of 111
    5.1490 -  4196.9668 [  8380]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
...
# Total for all samples = 713835.7990000012
```

- 128-Cores (condensed output):

```
=> Syscall: recvmsg/47 (duration in microseconds):
# NumSamples = 9120; Min = 5.20; Max = 53815.21
# Mean = 206.156264; Variance = 4907097.934233; SD = 2215.197042; Median 100.122500
# each ▌ represents a count of 121
    5.2000 -  5386.2010 [  9097]: ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
...
# Total for all samples = 1880145.1260000016
```

- I did not further investigate this more 2x increase.
- Could this be related to the additional statistics cost in getifaddrs()?
  - It’s not likely, see next section (≈ 6%)
- This needs further investigation!

`getifaddrs()` Statistics Gathering

- Glibc's getifaddrs() function also retrieves all interface statistics
  - Leads to unsesacry overhead for systems with a large number of CPU due to per-CPU IPv6 statistics.
  - Can be avoided with the RTEXT_FILTER_SKIP_STATS flag

- We aim to test the impact on **`getifaddrs()`** with 1024 interfaces.
  - We create 1024 veth interfaces and add them to OVS (see Kernel Interfaces Statistics Impact)
  - Add USDT probes around **`getifaddrs()`** in **`netdev_get_addrs()`**.
  - Measure for 5 minutes doing 10 revalidations per second.

- Results:

System	***`getifaddrs()`* runtime (ns)**
64-CPUs	3985.7
128-CPUs	4227.6
Delta(%)	*241.9(6.1%)*

- Next step would be to patch **`getifaddrs()`** in glibc to not gather the statistics.

Conclusion / Follow up

TO DO:

- Explore removing mutex in *`nl_dump_next()`*.
- Optimize the kmod's `sw_flow_stats` for faster combined stats gathering.
- Analyze `netdev_get_addrs()` delays in POD simulation.
- Check if unnecessary *`getifaddrs()`* stats cause delays; submit a glibc patch if needed.
- Asses of upcall-related impact measurements, particularly if ct() and nat() are involved.

Questions?

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.


linkedin.com/company/red-hat


youtube.com/user/RedHatVideos


facebook.com/redhatinc


twitter.com/RedHat

Red Hat