The CPU Cost of Networking on a Host

May 14, 2020

When evaluating networking for a host the focus is typically on latency, throughput or packets per second (pps) to see the maximum load a system can handle for a given configuration. While those are important and often telling metrics, results for such benchmarks do not tell you the impact processing those packets has on the workloads running on that system.

This post looks at the cost of networking in terms of CPU cycles stolen from processes running in a host.

Packet Processing in Linux

Linux will process a fair amount of packets in the context of whatever is running on the CPU at the moment the irq is handled. System accounting will attribute those CPU cycles to any process running at that moment even though that process is not doing any work on its behalf. For example, 'top' can show a process appears to be using 99+% cpu but in reality 60% of that time is spent processing packets meaning the process is really only get 40% of the CPU to make progress on its workload.

net_rx_action, the handler for network Rx traffic, *usually* runs really fast – like under 25 usecs[1] – dealing with up to 64 packets per napi instance (NIC and RPS) at a time before deferring to another softirq cycle. softirq cycles can be back to back, up to 10 times or 2 msec (see __do_softirq), before taking a break. If the softirq vector still has more work to do after the maximum number of loops or time is reached, it defers further work to the ksoftirqd thread for that CPU. When that happens the system is a bit more transparent about the networking overhead in the sense that CPU usage can be monitored (though with the assumption that it is packet handling versus other softirqs).

One way to see the above description is using perf:

```
sudo perf record -a \
        -e irq:irq_handler_entry,irq:irq_handler_exit \
        -e irq:softirq_entry --filter="vec == 3" \
        -e irq:softirq_exit --filter="vec == 3"  \
        -e napi:napi_poll \
        -- sleep 1

sudo perf script
```

The output is something like:

```
swapper     0 [005] 176146.491879: irq:irq_handler_entry: irq=152
name=mlx5_comp2@pci:0000:d8:00.0
swapper     0 [005] 176146.491880:  irq:irq_handler_exit: irq=152 ret=handled
swapper     0 [005] 176146.491880:     irq:softirq_entry: vec=3
[action=NET_RX]
swapper     0 [005] 176146.491942:        napi:napi_poll: napi poll on napi
struct 0xffff9d3d53863e88 for device eth0 work 64 budget 64
swapper     0 [005] 176146.491943:      irq:softirq_exit: vec=3
[action=NET_RX]
swapper     0 [005] 176146.491943:     irq:softirq_entry: vec=3
[action=NET_RX]
swapper     0 [005] 176146.491971:        napi:napi_poll: napi poll on napi
struct 0xffff9d3d53863e88 for device eth0 work 27 budget 64
swapper     0 [005] 176146.491971:      irq:softirq_exit: vec=3
[action=NET_RX]
swapper     0 [005] 176146.492200: irq:irq_handler_entry: irq=152
name=mlx5_comp2@pci:0000:d8:00.0
```

In this case the cpu is idle (hence swapper for the process), an irq fired for an Rx queue on CPU 5, softirq processing looped twice handling 64 packets and then 27 packets before exiting with the next irq firing 229 usec later and starting the loop again.

The above was recorded on an idle system. In general, any task can be running on the CPU in which case the above series of events plays out by interrupting that task, doing the irq/softirq dance and with system accounting attributing cycles to the interrupted process. Thus, processing packets is typically hidden from the usual CPU monitoring as it is done in the context of some random, victim process, so how do you view or quantify the time a process is interrupted handling packets? And how can you compare 2 different networking solutions to see which one is less disruptive to a workload?

With RSS, RPS, and flow steering, packet processing is usually distributed across cores, so the packet processing sequence describe above is all per-CPU. As packet rates increase (think 100,000 pps and up) the load means 1000's to 10,000's of packets are processed per second per cpu. Processing that many packets will inevitably have an impact on the workloads running on those systems.

Let's take a look at one way to see this impact.

Undo the Distributed Processing

First, let's undo the distributed processing by disabling RPS and installing flow rules to force the processing of all packets for a specific MAC address on a single, known CPU. My system has 2 nics enslaved to a bond in an 802.3ad configuration with the networking load targeted at a single virtual machine running in the host.

RPS is disabled on the 2 nics using

```
for d in eth0 eth1; do
    find /sys/class/net/${d}/queues -name rps_cpus |
    while read f; do
            echo 0 | sudo tee ${f}
    done
done
```

Next, add flow rules to push packets for the VM under test to a single CPU

```
DMAC=12:34:de:ad:ca:fe
sudo ethtool -N eth0 flow-type ether dst ${DMAC} action 2
sudo ethtool -N eth1 flow-type ether dst ${DMAC} action 2
```

Together, lack of RPS + flow rules ensure all packets destined to the VM are processed on the same CPU. You can use a command like ethq[3] to verify packets are directed to the expected queue and then map that queue to a CPU using /proc/interrupts. In my case queue 2 is handled on CPU 5.

openssl speed

I could use perf or a bpf program to track softirq entry and exit for network Rx, but that gets complicated quick, and the observation will definitely influence the results. A much simpler and more intuitive solution is to infer the networking overhead using a well known workload such as 'openssl speed' and look at how much CPU access it really gets versus is perceived to get (recognizing the squishiness of process accounting).

'openssl speed' is a nearly 100% userspace command and when pinned to a CPU will use all available cycles for that CPU for the duration of its tests. The command works by setting an alarm for a given interval (e.g., 10 seconds here for easy math), launches into its benchmark and then uses times() when the alarm fires as a way of checking how much CPU time it was actually given. From a syscall perspective it looks like this:

```
alarm(10)                               = 0
times({tms_utime=0, tms_stime=0, tms_cutime=0, tms_cstime=0}) = 1726601344
--- SIGALRM {si_signo=SIGALRM, si_code=SI_KERNEL} ---
rt_sigaction(SIGALRM, ...) = 0
rt_sigreturn({mask=[]}) = 2782545353
times({tms_utime=1000, tms_stime=0, tms_cutime=0, tms_cstime=0}) = 1726602344
```

so very few system calls between the alarm and checking the results of times(). With no/few interruptions the tms_utime will match the test time (10 seconds in this case).

Since it is is a pure userspace benchmark ANY system time that shows up in times() is overhead. openssl may be the process on the CPU, but the CPU is actually doing something else, like processing packets. For example:

```
alarm(10)                               = 0
times({tms_utime=0, tms_stime=0, tms_cutime=0, tms_cstime=0}) = 1726617896
--- SIGALRM {si_signo=SIGALRM, si_code=SI_KERNEL} ---
rt_sigaction(SIGALRM, ...) = 0
rt_sigreturn({mask=[]}) = 4079301579
times({tms_utime=178, tms_stime=571, tms_cutime=0, tms_cstime=0}) =
1726618896
```

shows that openssl was on the cpu for 7.49 seconds (178 + 571 in .01 increments), but 5.71 seconds of that time was in system time. Since openssl is not doing anything in the kernel, that 5.71 seconds is all overhead – time stolen from this process for “system needs.”

Using openssl to Infer Networking Overhead

With an understanding of how 'openssl speed' works, let's look at a near idle server:

```
$ taskset -c 5 openssl speed -seconds 10 aes-256-cbc >/dev/null
Doing aes-256 cbc for 10s on 16 size blocks: 66675623 aes-256 cbc's in 9.99s
Doing aes-256 cbc for 10s on 64 size blocks: 18096647 aes-256 cbc's in 10.00s
Doing aes-256 cbc for 10s on 256 size blocks: 4607752 aes-256 cbc's in 10.00s
Doing aes-256 cbc for 10s on 1024 size blocks: 1162429 aes-256 cbc's in
10.00s
Doing aes-256 cbc for 10s on 8192 size blocks: 145251 aes-256 cbc's in 10.00s
Doing aes-256 cbc for 10s on 16384 size blocks: 72831 aes-256 cbc's in 10.00s
```

so in this case openssl reports 9.99 to 10.00 seconds of run time for each of the block sizes confirming no contention for the CPU. Let's add network load, netperf TCP_STREAM from 2 sources, and re-do the test:

```
$ taskset -c 5 openssl speed -seconds 10 aes-256-cbc >/dev/null
Doing aes-256 cbc for 10s on 16 size blocks: 12061658 aes-256 cbc's in 1.96s
Doing aes-256 cbc for 10s on 64 size blocks: 3457491 aes-256 cbc's in 2.10s
Doing aes-256 cbc for 10s on 256 size blocks: 893939 aes-256 cbc's in 2.01s
Doing aes-256 cbc for 10s on 1024 size blocks: 201756 aes-256 cbc's in 1.86s
Doing aes-256 cbc for 10s on 8192 size blocks: 25117 aes-256 cbc's in 1.78s
Doing aes-256 cbc for 10s on 16384 size blocks: 13859 aes-256 cbc's in 1.89s
```

Much different outcome. Each block size test wants to run for 10 seconds, but times() is reporting the actual user time to be between 1.78 and 2.10 seconds. Thus, the other 7.9 to 8.22 seconds was spent processing packets – either in the context of openssl or via ksoftirqd.

Looking at top for the previous openssl run:

```
 PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
P
 8180 libvirt+  20   0 33.269g 1.649g 1.565g S 279.9  0.9  18:57.81 qemu-
system-x86     75
 8374 root      20   0       0      0      0 R  99.4  0.0   2:57.97
vhost-8180         89
 1684 dahern    20   0   17112   4400   3892 R  73.6  0.0   0:09.91 openssl
5
   38 root      20   0       0      0      0 R  26.2  0.0   0:31.86
ksoftirqd/5         5
```

one would think openssl is using ~73% of cpu 5 with ksoftirqd taking the rest but in reality so many packets are getting processed in the context of openssl that it is only effectively getting 18-21% time on the cpu to make progress on its workload.

If I drop the network load to just 1 stream, openssl appears to be running at 99% CPU:

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
P
 8180 libvirt+  20   0 33.269g 1.722g 1.637g S 325.1  0.9 166:38.12 qemu-
system-x86     29
44218 dahern    20   0   17112   4488   3996 R  99.2  0.0   0:28.55 openssl
5
 8374 root      20   0       0      0      0 R  64.7  0.0  60:40.50
```

```
vhost-8180          55
   38 root       20   0       0      0      0 S   1.0  0.0   4:51.98
ksoftirqd/5          5
```

but openssl reports ~4 seconds of userspace time:

```
Doing aes-256 cbc for 10s on 16 size blocks: 26596388 aes-256 cbc's in 4.01s
Doing aes-256 cbc for 10s on 64 size blocks: 7137481 aes-256 cbc's in 4.14s
Doing aes-256 cbc for 10s on 256 size blocks: 1844565 aes-256 cbc's in 4.31s
Doing aes-256 cbc for 10s on 1024 size blocks: 472687 aes-256 cbc's in 4.28s
Doing aes-256 cbc for 10s on 8192 size blocks: 59001 aes-256 cbc's in 4.46s
Doing aes-256 cbc for 10s on 16384 size blocks: 28569 aes-256 cbc's in 4.16s
```

Again, monitoring tools show a lot of CPU access, but reality is much different with 55-80% of the CPU spent processing packets. The throughput numbers look great (22+Gbps for a 25G link), but the impact on processes is huge.

In this example, the process robbed of CPU cycles is a silly benchmark. On a fully populated host the interrupted process can be anything – virtual cpus for a VM, emulator threads for the VM, vhost threads for the VM, or host level system processes with varying degrees of impact on performance of those processes and the system.

Up Next

This post is the basis for a follow up post where I will discuss a comparison of the overhead of "full stack" on a VM host to "XDP".

[1] Measured using ebpf program on entry and exit. See net_rx_action in https://github.com/dsahern/bpf-progs

[2] Assuming no bugs in the networking stack and driver. I have examined systems where net_rx_action takes well over 20,000 usec to process less than 64 packets due to a combination of bugs in the NIC driver (ARFS path) and OVS (thundering herd wakeup).

[3] https://github.com/isc-projects/ethq