Network Performance in the Linux Kernel

Maxime Chevallier

maxime.chevallier@bootlin.com

© Copyright 2004-2021, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!
Maxime Chevallier

Linux kernel engineer at Bootlin.
- Linux kernel and driver development, system integration, boot time optimization, consulting...
- Embedded Linux, Linux driver development, Yocto Project & OpenEmbedded and Buildroot training, with materials freely available under a Creative Commons license.
  - https://bootlin.com

Contributions:
- Worked on network (MAC, PHY, switch) engines.
- Contributed to the Marvell EBU SoCs upstream support.
- Worked on Rockchip’s Camera interface and Techwell’s TW9900 decoder.
Preamble - goals

- Follow the path of packets through the Hardware and Software stack
- Understand the features of modern NICs
- Discover what the Linux Kernel implements to gain performances
- Go through the various offloadings available
The path of a packet

Maxime Chevallier
maxime.chevallier@bootlin.com
The Hardware

- **Link Partner**: The other side of the cable
- **Connector**: 8P8C (RJ45), SFP, etc.
- **Media**: Copper, Fiber, Radio
- **PHY**: Converts media-dependent signals into standard data

![Diagram showing hardware components and connectivity](image-url)
The NIC

- **Network Interface Controller**
- Sometimes embed a PHY (PCIe networking card)
- The MAC : Handles L2 protocol, transfers data to the CPU
Frame reception

- The MAC received data and writes it to RAM using DMA.
- A descriptor is created.
- Its address is put in a queue.
An interrupt is fired

One CPU core will handle the interrupt
The Interrupt handler acknowledges the interrupt

The packet is processed in `softirq` context

The next frame can be received in parallel
In the NIC driver

- The CPU: Processes L3 (packets) and above, up to the application
- The Interrupt Handler does very basic work, and masks interrupts
- NAPI is used to schedule the processing in batches
- Subsequent frames are also dequeued
- NAPI stops dequeuing once:
  - The budget is expired (release the CPU to the scheduler)
  - The queue is empty
- NAPI re-enables interrupts
  - This avoids having one interrupt per frame
In the kernel networking stack

- The PCAP hook is called, then the TC hook
- The header is unpacked, to decide if:
  - The packet is forwarded
  - The packet is dropped
  - The packet is passed to a socket
- The in-kernel data path is heavily optimized...
- ...But still requires some processing power at very high speeds
Traffic Spreading and Steering

Maxime Chevallier
maxime.chevallier@bootlin.com
Most modern systems have multi-core CPUs

Modern NICs have multiple RX/TX queues (rxq/txq)

Interrupted CPU does all the packet processing
  ▶ If the interrupt always goes to the same core...
  ▶ ...the other ones will stay unused

Hardware and Software techniques exists to scale processing across CPUs
Scaling

Goal: Spread packet across CPU cores

- We can’t randomly assign packets to CPUs
- Ordering must be preserved
- Memory domains should be taken into account (L1/L2 caches, NUMA nodes)
- We need to spread packets per-flow

Kernel documentation: Documentation/networking/scaling.rst
**N-tuple Flows**

**Flow**: Packets from the same emitter, for the same consumer

- Flows are identified by data extracted from the headers
- L3 flow: Source and Destination IP addresses → 2-tuple
- L4 flow: src/dst IP + Proto + src/dst ports → 5-tuple

```
<table>
<thead>
<tr>
<th>L2</th>
<th>MAC SA</th>
<th>MAC DA</th>
</tr>
</thead>
<tbody>
<tr>
<td>vlan</td>
<td>ethtype</td>
<td></td>
</tr>
<tr>
<td>ver</td>
<td>ToS len</td>
<td>id</td>
</tr>
<tr>
<td>Proto</td>
<td>checksum</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L3</th>
<th>IP SA</th>
<th>IP DA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC port</td>
<td>DST port</td>
<td></td>
</tr>
</tbody>
</table>

| L4 | len, chksum, etc. |

| Payload |
```

```
<table>
<thead>
<tr>
<th>L2</th>
<th>MAC SA</th>
<th>MAC DA</th>
</tr>
</thead>
<tbody>
<tr>
<td>vlan</td>
<td>ethtype</td>
<td></td>
</tr>
<tr>
<td>ver</td>
<td>ToS len</td>
<td>id</td>
</tr>
<tr>
<td>Proto</td>
<td>checksum</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L3</th>
<th>IP SA</th>
<th>IP DA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC port</td>
<td>DST port</td>
<td></td>
</tr>
</tbody>
</table>

| L4 | len, chksum, etc. |

| Payload |
```

2-tuple

5-tuple
N-tuple Flows

- Other flows can be interesting:
  - Vlan-based flows
  - Destination MAC-based flows
- Most often, these tuples are hashed prior to being used
- Allows to build smaller flow tables
- Advanced NICs are able to keep track of a high number of flows

```
key = cpu_id
```

```
Hash(key) % N
```

```
proto  IP SA  ...
```

```
L2
---
<table>
<thead>
<tr>
<th>MAC SA</th>
<th>MAC DA</th>
</tr>
</thead>
<tbody>
<tr>
<td>vlan</td>
<td>ethtype</td>
</tr>
<tr>
<td>ver</td>
<td>ToS</td>
</tr>
<tr>
<td>len</td>
<td>id</td>
</tr>
</tbody>
</table>

```

```
L3
---
| Proto  | checksum|
| IP SA  | IP DA   |
| SRC port | DST port |
| len, chksum, etc. |

```

```
L4
---
| Payload |
```

---

Kernel, drivers and embedded Linux - Development, consulting, training and support - https://bootlin.com
- The interrupted CPU schedules processing on other CPUs
- Key data is extracted from the Headers and hashed
  - The Hash can be computed by the hardware
  - It is then passed in the descriptor
- The CPU is chosen by masking out the first bits of the hash
Kernel needs to be built with `CONFIG_RPS`

The set of CPUs used depends on the `rxq` the frame arrives on:

- `echo 0x03 > /sys/class/net/eth0/queues/rx-0/rps_cpus`
- `echo 0x0c > /sys/class/net/eth0/queues/rx-1/rps_cpus`

  → Traffic from `rxq 0` is spread on CPUs 0 and 1
  → Traffic from `rxq 1` is spread on CPUs 2 and 3

Very useful for NICs with fewer `rxqs` than CPU cores!
RSS : Receive Side Scaling

- Offloaded version of RPS
- The NIC is configured to extract the header data and compute the Hash
- The CPU is chosen by means of an Indirection Table
- The NIC actually enqueues the packet into one of its queues
- The interrupt directly comes to the correct CPU!
RSS Tables

- Tables within the NIC
- Associates hashes to queues
- Tables commonly have more entries than queues
  - e.g. 128 entries for 4 queues
- Filling the table allows affecting weights to each queue

```
0x00: 0 0 0 0 0 0 0 0
0x08: 0 0 0 0 0 0 0 0
0x10: 1 1 1 1 1 1 1 1
0x18: 2 2 2 2 3 3 3 3
```

- `rxq 0` has weight 4
- `rxq 1` has weight 2
- `rxq 2` and `3` have weight 1
Using RSS

- RSS is configured through ethtool
- Enabling RSS: `ethtool -K eth0 rx-hashing on`
- Configuring the indirection table:
  `ethtool -X eth0 weight 1 2 2 1`
- Dumping the indirection table: `ethtool -x eth0`
- Configuring the hashed fields:
  `ethtool -N eth0 rx-flow-hash tcp4 sdn`
  - see man ethtool(8) for the meaning of each flow type

```
ethtool -X eth0 equal 4
ethtool -N eth0 rx-flow-hash tcp4 sdn
ethtool -N eth0 rx-flow-hash udp4 sdn
ethtool -K eth0 rx-hashing on
```

→ Increased IP forwarding speed by a factor of 3 on the MacchiatoBin
RFS: Receive Flow Steering

- RPS and RSS don’t care about who consumes which flow
- This might be bad for cache locality
  - What if RPS/RSS sends a flow to CPU 1...
  - ...but the consumer process lives on CPU 2?
- RFS tracks the flows and their consumers
- Internally keeps a table associating flows to consumers/CPUs
- Updates indirection for a flow when the consumer migrates

1. httpd lives on CPU 0
2. RFS steers TCP traffic to port 80 onto CPU 0
3. httpd is migrated to CPU 1
4. RFS updates the flow table
5. TCP to port 80 traffic now goes to CPU 1!
Using RFS

- Internally, a **flow table** associates flow hashes to CPUs
- User indicates the size of the table
  - `echo 32768 > /proc/sys/net/core/rps_sock_flow_entries`
  - `echo 4096 > /sys/class/net/eth0/queues/rx-0/rps_flow_cnt`
  - `echo 4096 > /sys/class/net/eth0/queues/rx-1/rps_flow_cnt`
  - ...  
- We configure \( \frac{32768}{N(rxs)} \) in each queue
- These values are recommended in the Kernel Documentation
aRFS : Accelerated Receive Flow Steering

- Advanced NICs can steer packets to \textit{rxqs} in Hardware
- aRFS asks the driver to configure steering rules for each flow
- Rules are updated upon migration of the consumer
  - Packets always come to the right CPU!
  - Kernel handles outstanding packets upon migration
- Needs support in HW, and a specific implementation in the driver
- The driver determines how to build the steering rule (n-tuple)
Kernel needs to be built with `CONFIG_RFS_ACCEL`

Enable N-tuple filtering offloading:
```
ethtool -K eth0 ntuple on
```

The NIC and the driver needs to support aRFS
Flow Steering : Ethtool and TC flower

- Manually steering flows can be interesting for proper resource assignment
- This is also helpful to dedicate queues to flows, e.g. AF_XDP
- Two interfaces exists: tc flower and ethtool
  - Internally, both ethtool and tc interfaces are being merged...
  - ... But for now the 2 methods coexist and can conflict
- We insert **steering rules** in the NIC, with priorities
- Rules associate:
  - Flow types: TCP4, UDP6, IP4, ether, etc.
  - Filters: src-ip, proto, vlan, dst-port, etc.
  - Actions: Target rxq, drop, RSS context
  - Location: Priority of the rule
Using tc flower and ethtool rxnfc

ethtool examples

- `ethtool -K eth0 ntuple on`
- `ethtool -N eth0 flow-type udp4 dst-port 1234 action 2 loc 0`
  - Steer IPv4 UDP traffic for port 1234 to rxq 2
- `ethtool -N eth0 flow-type udp4 action -1 loc 1`
  - Drop all UDP IPv4 traffic (except for port 1234)

TC flower example

- `ethtool -K eth0 hw-tc-offload on`
- `tc qdisc add dev eth0 ingress`
- `tc flower protocol ip parent ffff: flower ip_proto tcp dst_port 80 action drop`
  - Drop all IPv4 TCP traffic for port 80
- `tc flower falls back to software filtering if needed`
RSS contexts

- Flows can also be steered to multiple queues at once
- RSS is then used to spread traffic across queues
- This is achieved through RSS contexts
- An RSS context is simply an indirection table
- An RSS context is created with ethtool:
  - `ethtool -X eth0 equal 4 context new`
- The RSS context is used as a destination for the flow:
  - `ethtool -N eth0 flow-type udp4 dst-port 1234 context 1 loc 0`
Upon transmitting packets, the driver executes *completion* code.

Transmitting using a single CPU can also lead to cache misses.

XPS is used to select which *txq* to use for packet sending.

We can assign *txqs* to *CPUs*.

- The *txq* is chosen according to the CPU the sender lives on.

We can also assign *txqs* to *rxqs*.

- Make sure that we use the same CPU for RX and TX.

The *NIC* driver assigns *txqs* to CPUs.
Using XPS

▶ Per-CPU mapping:
  ▶ echo 0x01 > /sys/class/net/eth0/queues/tx-0/xps_cpus
  ▶ echo 0x02 > /sys/class/net/eth0/queues/tx-1/xps_cpus
  ▶ Assign txq 0 to CPU 0
  ▶ Assign txq 1 to CPU 1

▶ Per-rxq mapping:
  ▶ echo 0x01 > /sys/class/net/eth0/queues/tx-0/xps_rxqs
  ▶ echo 0x02 > /sys/class/net/eth0/queues/tx-1/xps_rxqs
  ▶ Assign txq 0 to rxq 0
  ▶ Assign txq 1 to rxq 1
Other offloading

Maxime Chevallier
maxime.chevallier@bootlin.com
Checksumming

- IPv4 and IPv6 include a checksum in the header
- NICs can compute checksums on the fly in TX mode
- The Kernel leaves the checksum fields empty
- tcpdump will show egress packets with a wrong checksum !!
Filtering

- Some NICs are capable of early dropping and filtering
- Frames are dropped by the NIC, no interrupt is ever fired
- MAC filtering:
  - Drop frames with an unknown MAC address
  - The NIC keeps information about multicast domains
  - The NIC must also keep an updated list of unicast addresses
  - MAC Vlan allows attaching multiple addresses to one NIC
- VLAN filtering:
  - Drop frames for unknown VLANs
  - The NIC keeps track of VLANs attached to the interface
  - `ethtool -K eth0 rx-vlan-filter on`
Data insertion and segmentation

- Some NICs can also insert the VLAN tag on the fly
  - `ethtool -K eth0 txvlan on`
    - The NIC will insert the VLAN Tag automatically
  - `ethtool -K eth0 rxvlan on`
    - The NIC will strip the VLAN tag
    - The VLAN tag will be in the descriptor

- Some NICs can also deal with packet segmentation
  - `ethtool -K eth0 tso on`
    - Offload TCP segmentation, the NIC will generate segments
  - `ethtool -K eth0 ufo on`
    - Offload UDP fragmentation, the NIC will generate fragments
XDP

Maxime Chevallier
maxime.chevallier@bootlin.com
Principle

- Execute a BPF program from within the NIC driver
- Executed as early as possible, for fast decision making
- Can be used to Pass, Drop or Redirect frames
- Also used for fine-grained statistics

**BPF**

- Berkley Packet Filter
- Programming language that can be formally verified
- Designed to write filtering rules
- Lots of hooks in the Networking Stack, XDP being the earliest
AF_XDP

- Uses a combination of XDP and flow steering
- Response to DPDK: Userspace does the full packet processing
- Allows for heavily optimized and customized processing
- Special sockets that will directly receive raw buffers
- Thanks to XDP, we can select only part of the traffic for AF_XDP
- The kernel stack is therefore not entirely bypassed...
- ...and this is a fully upstream solution!
In the Kernel source code:
Documentation/networking/scaling.rst

RedHat tuning guide:
That’s it

Thank you!