Prototype Kernel Documentation

Release 0.0.1

Jesper Dangaard Brouer

Jun 23, 2017

Contents

This project and GitHub repository is meant for speeding up Linux Kernel development work, this also includes Documentation. The directory layout tries to keep close to the Kernel directory layout. This helps when/if upstreaming the work.

Contents:

CHAPTER 1

Documentation

This documentation is available at: prototype-kernel.readthedocs.io

Files in this Documentation/ directory is (like the kernel) based on reStructuredText files and Sphinx can be used for generating pretty documentation. Just like this documentation is being auto-generated on Read The Docs.

Compiling

To generate pretty Sphinx documentation locally simply run

```
make html
```

The generated output will be located in `_build/html/index.html`.

CHAPTER 2

Prototype Kernel

This documentation is for how to use the prototype-kernel project itself.

XDP and eBPF

This github repository also contains samples for XDP and eBPF in the directory samples/bpf/. The build process is different. Simply run `make` in the directory. Also see *XDP/eBPF build environment*.

Prototype Kernel own documentation

The prototype-kernel project is meant for compiling kernel modules outside the normal kernel git tree, but still using the kernels make system.

The purpose is getting a separate git development tree for developing and refining your kernel module or Documentation over time, before pushing it upstream for the Linux Kernel.

Contents:

Prototype Kernel build process

In the `kernel/` directory we try to keep close to the kernel directory layout, in the hopes that it will make it easier, when posting/proposing these changes upstream.

Note: It is a pre-requisite that you have a developement kernel tree available for compiling against (or install your distributions kernel-devel package).

Compiling modules

To compile your modules simply type `make` in the `kernel` directory.

The `Makefile` tries to detect the kernel directory to compile against by following the running kernels build symlink in:

```
/lib/modules/`uname -r`/build/
```

To compile against another (specific) kernel tree use:

```
make kbuilddir=~/git/kernel/net-next/
```

Notice look in the `Kbuild` files, they define and control which modules are compiled, also see the `.config` file.

Push to remote host

Q: Want to compile locally and push the binary modules to a remote host. A: Yes, this is supported.

The Makefile target "push_remote" uploads the kernel module to a remote host. (You need to setup SSH-keys to SSH allow root logins.)

Usage example:

```
make push_remote kbuilddir=~/git/kernel/net-next/ HOST=192.168.122.49
```

If you want to run this manually call the script directly:

```
./scripts/push_remote.sh 192.168.122.49
```

Enable/disable modules

It can be practical to allow manual enable/disable of which modules are getting build. This is supported by locally adjusting .config. On first run the content is based on config.default.

This feature is useful when developing against API's that have not been included the mainline kernel yet. See CONFIG_SLAB_BULK_API=m for an example.

CHAPTER 3

Linux Networking Subsystem

This is the top-level documentation for the Linux Networking subsystem.

Contents:

XDP - eXpress Data Path

This is the top-level XDP documentation tree.

Contents:

Introduction

What is XDP?

XDP or eXpress Data Path provides a high performance, programmable network data path in the Linux kernel. XDP provides bare metal packet processing at the lowest point in the software stack. Much of the huge speed gain comes from processing RX packet-pages directly out of drivers RX ring queue, before any allocations of meta-data structures like SKBs occurs.

The IO Visor Project have an introduction to XDP.

Presentations

List of XDP focused presentations:

- March 2016 - Initial presentation by Facebook (Tom and Alexei)
- July 2016 - IO visor (Brenden Blanco)
- September 2016 - Intro and use-case, Red Hat Inc. (Jesper Brouer)
- April 2017 - Keynote NetDevconf 2.1: XDP Mythbusters

- April 2017 - XDP/eBPF tutorial: XDP for the Rest of Us
- April 2017 - Facebook Droplet
- April 2017 - CloudFlare integrating XDP

Historically the Network Performance BoF at NetDev 1.1 (Feb 2016) was the first presentation to propose the idea of processing RX packet-pages directly out of the driver RX ring queue.

Press coverage

List of press coverage:

- April 2016 - LWN.net covered the very early patches

Related resources

List of related presentations or write-ups:

- (Juli 2016): Next Steps for Linux Network Stack (Video)
- (Juli 2016): CETH Common Ethernet Driver Framework (Huawei)
- (Aug 2016): What Can BPF Do For You (LinuxCon)
- (Sep 2016): Dive into BPF: a list of reading material
- (Oct 2016): XDP in OpenStack (video) for DDoS protection
- (Oct 2016): NetDev 1.2 video by David Miller
- (April 2017): BPF and XDP Reference Guide: Cilium developer's guide

Disclaimer

XDP is not for every use-case.

Important to understand

It is important to understand that the XDP speed gains comes at a cost of loss of generalization and fairness.

XDP does not provide fairness. There is no buffering (qdisc) layer to absorb traffic bursts when the TX device is too slow, packets will simply be dropped. Don't use XDP in situations where the RX device is faster than the TX device, as there is no back-pressure to save the packet from being dropped. There is no qdisc layer or BQL (Byte Queue Limit) to save you from introducing massive bufferbloat.

Using XDP is about specialization. Crafting a solution towards a very specialized purpose, that will require selecting and dimensioning the appropriate hardware. Using XDP requires understanding the dangers and pitfalls, that come from bypassing large parts of the kernel network stack code base, which is there for good reasons.

That said, XDP can be the right solution for some use-cases, and can yield huge (orders of magnitude) performance improvements, by allowing this kind of specialization.

Design

XDP is designed for high performance. It uses known techniques and applies selective constraints to achieve performance goals. XDP is also designed for programmability. New functionality can be implemented on the fly without needing kernel modification

Contents:

Overall design

Requirements defined in document *Requirements*.

Programmability

XDP is designed for programmability.

Users want programmability as close as possible to the device hardware, to reap the performance gains, but they also want portability. The purpose of XDP is making such programs portable across multiple devices and vendors.

(It is even imagined that XDP programs should be able to run in user space, either for simulation purposes or combined with other raw packet data-plane frameworks like netmap or DPDK).

It is expected that some HW vendors will take steps towards offloading XDP programs into their hardware. It is fine if they compete on this to sell more hardware. It is no different from producing the fastest chip. XDP also encourages innovation for new HW features, but when extending XDP programs with a new hardware feature (e.g. which only a single vendor supports), this must be expressed within the XDP API as a capability or feature (see section *Capabilities negotiation*). This functions as a common capabilities API from which vendors can choose what to implement (based on customer demand).

Capabilities negotiation

Warning: This interface is missing in the implementation

XDP has hooks and feature dependencies in the device drivers. Planning for extendability, not all device drivers will necessarily support all of the future features of XDP, and new feature adoption in device drivers will occur at different development rates.

Thus, there is a need for the device driver to express what XDP capabilities or features it provides.

When attaching/loading an XDP program into the kernel, a feature or capabilities negotiation should be conducted. This implies that an XDP program needs to express what features it wants to use.

If an XDP program being loaded requests features that the given device driver does not support, the program load should simply be rejected.

Note: I'm undecided on whether to have an query interface, because users could just use the regular load-interface to probe for supported options. The downside of probing is the issues SElinux runs into, of false alarms, when glibc tries to probe for capabilities.

Implementation issue

The current implementation is missing this interface. Worse, the two actions *XDP_DROP* and *XDP_TX* should have been expressed as two different capabilities, because XDP_TX requires more changes to the device driver than a simple drop like XDP_DROP.

One can (easily) imagine that an older driver only wants to implement the XDP_DROP facility. The reason is that XDP_TX would require changing too much driver code, which is a concern for an old, stable and time-proven driver.

Data plane split

Requirements

Driver RX hook

Gives us access to packet-data payload before allocating any meta-data structures, like SKBs. This is key to performance, as it allows processing RX "packet-pages" directly out of the driver's RX ring queue.

Early drop

Early drop is key for the DoS (Denial of Service) mitigation use-cases. It builds upon a principle of spending/investing as few CPU cycles as possible on a packet that will get dropped anyhow.

Doing this "inline", before delivery to the normal network stack, has the advantage that a packet that *does* need delivery to the normal network stack can still get all the features and benefits as before; there is thus no need to deploy a bypass facility merely to re-inject "good" packets into the stack again.

Write access to packet data

XDP needs the ability to modify packet data. This is unfortunately often difficult to obtain, as it requires fundamental changes to the driver's memory model.

Unfortunately most drivers don't have "writable" packet data as default. This is due to a workaround for performance bottlenecks in both the page-allocator and DMA APIs, which has the side-effect of necessitating read-only packet pages.

Instead, most drivers (currently) allocate both a SKB and a writable memory buffer, in which to copy ("linearise") the packet headers, and also store `skb_shared_info`. Then the remaining payload (pointing past the headers just copied) is attached as (read-only) paged data.

Header push and pop

The ability to push (add) or pop (remove) packet headers indirectly depends on write access to packet-data. (One could argue that a pure pop could be implemented by only adjusting the payload offset, thus not needing write access).

This requirement goes hand-in-hand with tunnel encapsulation or decapsulation. It is also relevant for e.g adding a VLAN header, as needed by the *Use-case: DDoS scrubber* in order to work around the *XDP_TX* single NIC limitation.

This requirement implies the ability to adjust the packet-data start offset/pointer and packet length. This requires additional data to be returned.

This also has implications for how much headroom drivers should reserve in the SKB.

Page per packet

On RX many NIC drivers split up a memory page, to share it for multiple packets, in-order to conserve memory. Doing so complicates handling and accounting of these memory pages, which affects performance. Particularly the extra atomic refcnt handling needed for the page can hurt performance.

XDP defines upfront a memory model where there is only one packet per page. This simplifies page handling and open up for future extensions.

This requirement also (upfront) result in choosing not to support things like, jumbo-frames, LRO and generally packets split over multiple pages.

In the future, this strict memory model might be relaxed, but for now it is a strict requirement. With a more flexible *Capabilities negotiation* it might be possible to negotiate another memory model. Given some specific XDP use-case might not require this strict memory model.

Packet forwarding

Implementing a router/forwarding data plane is DPDK's prime example for demonstrating superior performance. For the sheer ability to compare against DPDK, XDP also needs a forwarding capability.

RX bulking

Implementation

This document section is primarily for coordinating the XDP infrastructure **developers**.

Keeping track of *Missing Features* and details about suboptimal implementations that need to be looked at.

Contents:

XDP actions

XDP_PASS

XDP_PASS means the XDP program chose to pass the packet to the normal network stack for processing. Note that the XDP program is allowed to have modified the packet-data.

XDP_DROP

XDP_DROP is perhaps the simplest and fastest action. It simply instructs the driver to drop the packet. Given this action happens at the earliest RX stage in the driver, dropping a packet simply implies recycling it back-into the RX ring queue it just "arrived" on. There is simply no faster way to drop a packet. This comes close to a driver hardware test feature.

XDP_TX

The XDP_TX action result in TX bouncing the received packet-page back out the same NIC it arrived on. This is usually combined with modifying the packet contents before returning action XDP_TX.

The XDP_TX feature can be used for implementing a special kind of one-legged Load-Balancer as described in *Use-case: Load Balancer*.

XDP_ABORTED

The XDP_ABORTED action is not something a functional program should ever use as a return code. This return code is something an `eBPF` program returns in case of an eBPF program error, e.g. division by zero. For this reason XDP_ABORTED will always be the value zero.

This XDP_ABORTED action results in the packet getting dropped.

For how to troubleshoot this kind of unlikely error event, see the section *Troubleshooting and Monitoring*.

Fall-through

There must also be a fall-through `default:` case, which is hit if the program returns an unknown action code (e.g. future action this driver does not support).

These unknown return codes will result in packet drop.

See the section *Troubleshooting and Monitoring* for how to catch these kind of situations.

Code example

The basic action code block the driver use, is simply a switch-case statement as below.

```
switch (action) {
        case XDP_PASS:
                break; /* Normal netstack handling */
        case XDP_TX:
                if (driver_xmit(dev, page, length) == NETDEV_TX_OK)
                        goto consumed;
                goto xdp_drop; /* Drop on xmit failure */
        default:
                bpf_warn_invalid_xdp_action(action);
        case XDP_ABORTED:
        case XDP_DROP:
xdp_drop:
                if (driver_recycle(page, ring))
                        goto consumed;
                goto next; /* Drop */
        }
}
```

Warning: It is still undecided whether the `action` code needs to be partitioned into opcodes, with some of the upper bits used as values for the given opcode. This can be extended later.

Userspace API

Warning: The userspace API specification should have been defined properly before code was accepted upstream. Concerns have been raised about the current API upstream. Users should expect this first API attempt will need adjustments; this cannot be considered a stable API yet.

Most importantly, capabilities negotiation is missing; see *Capabilities negotiation*.

Planning for API extension

The kernel documentation about syscalls have some good considerations when designing an extendable API, and Michael Kerrisk also have some entertaining API examples.

Note: With XDP_FLAGS in commit 85de8576a0b1 (Daniel) prepared add/replace/delete logic for XDP programs.

Struct xdp_prog

Currently (4.8-rc6) the XDP program is simply a bpf_prog pointer. While this is good for simplicity, it limits extendability for upcoming features.

Maybe we should introduce a new `struct xdp_prog` that can carry information related to the XDP program. Notice this approach does not affect performance (tested and benchmarked), because the extra dereference for the eBPF program only happens once per 64 packets in the poll function.

The features that need this are:

- Multi-port TX: Need to know own port index and port lookup table.
- XDP program per RX queue: Need setup info about program type, global or specific, due to program-replacement semantics.
- Capabilities negotiation: Need to store information about features program wants to use, in order to validate this.

Todo

How kernel devel works: This new `struct xdp_prog` feature cannot go into the kernel before one of the three users of the struct is also implemented. (Note, Jesper has implemented this struct change and has even benchmarked that it does not hurt performance).

XDP meta-data

The `struct xdp_md` carry XDP meta-data ("_md"). It is still extensible because it has a internal BPF insn rewriter.

Troubleshooting and Monitoring

Users need the ability to both monitor and troubleshoot an XDP program; particularly so in case of error events like *XDP_ABORTED*, and in case an XDP program starts to return invalid and unsupported action codes (caught by the *Fall-through*).

Note: Daniel choose to implement this as tracepoints. See commit: a67edbf4fb6d ("bpf: add initial bpf tracepoints") https://git.kernel.org/davem/net-next/c/a67edbf4fb6d Scheduled for kernel 4.11.

Warning: The current (4.8-rc6) implementation is not optimal in this area. In the *Fall-through* case, the packet is dropped and a warning is generated **only once** about the invalid XDP program action code, by calling: bpf_warn_invalid_xdp_action(action_code);

The facilities and behavior need to be improved in this area.

Two options are on the table currently:

- Counters.

  Simply add counters to track these events. This allows admins and monitoring tools to catch and count these events. This does require standardizing these counters to help monitor tools.

- Tracepoints.

  Another option is adding tracepoints to these situations. These are much more flexible than counters. The downside is that these error events might never be caught, if the tracepoint isn't active.

An important design consideration is that the monitor facility must not be too expensive to execute, even though events like *XDP_ABORTED* and *Fall-through* should normally be very rare. This is because an external attacker (given the DDoS uses-cases) might find a way to trigger these events, which would then serve as an attack vector against XDP.

Missing Features

Record missing implementation features here.

Missing: Push/pop headers

Requirement defined here: *Header push and pop*.

Needed by *Use-case: DDoS scrubber*

Initial support for XDP head adjustment added to net-next in this commit: https://git.kernel.org/davem/net-next/c/293bfa9b486

Initial support only covers driver mlx4.

Todo

Update document once feature is available in a kernel release. Plus, keep track of drivers supporting this feature.

Todo

Create new section under *Userspace API* that describe howto use this and point to sample programs.

The eBPF program gets a new helper function called: `bpf_xdp_adjust_head`

Missing: Multi-port TX

Missing: Capabilities negotiation

See: *Capabilities negotiation*

Missing: XDP program per RX queue

Changes to the user space API are needed to add this feature.

Missing: Cache prefetching

Drivers

XDP depends on drivers implementing the RX hook and set-up API. Adding driver support is fairly easy, unless it requires changing the driver's memory model (which is often the case).

Mellanox: mlx4

The first driver implementing XDP were the Mellanox `mlx4` driver. The corresponding NIC is called ConnectX-3 and ConnectX-3 pro. These NICs run Ethernet at 10Gbit/s and 40Gbit/s.

Mellanox: mlx5

The Mellanox driver `mlx5` support XDP since kernel v4.9, but kernel v4.10 is recommended as some minor fixes got applied.

These NICs run Ethernet at 10G, 25G, 40G, 50G and 100Gbit/s. They are called ConnectX-4 and ConnectX-4-Lx (Lx is limited to max 50G or 2x 25G).

Netronome: nfp

Driver: nfp Kernel release: v4.10

virtio-net

Driver: virtio-net Kernel release: v4.10

Cavium/Qlogic: qede

Driver: qede Kernel release: v4.10

Cavium: thunder

Driver: thunder/nicvf

- Kernel release: v4.12

Broadcom: bnxt

Driver: bnxt

- Kernel release: v4.12

Intel: ixgbe

Driver: ixgbe

- Kernel release: v4.12

Use-cases

XDP use-cases; some are only proposals.

Contents:

Use-case: DDoS

DDoS protection was the primary use-case XDP was born out of. CloudFlare presented their DDoS use-case at the Network Performance BoF at NetDev 1.1, which convinced many Kernel developers that this was something that needed to be solved.

End-host protection

When a server is under DoS (Denial-of-Service) attack, the attacker is trying to use as many resource on the server as possible, in order to not leave processing time to service the legitimate users.

Owing to XDP running so early in the software stack, there is almost no processing cost associated with dropping a packet. This makes it a viable option to load a XDP program directly on the server, as filtering out bad/attacker traffic (this early) frees up processing resources.

As XDP is still part of the Linux network stack, packets that "pass" the XDP filter still have all features for further filtering that the kernel normally provides. It works in concert with the regular network stack, rather than trying to by-pass it.

Use-case: DDoS scrubber

Version 0.2

Status Proposal, need some new XDP features

This document investigates whether XDP can be used for implementing a machine that does traffic scrubbing at the edge of the network.

DDoS volume attacks

This idea/use-case comes from a customer. They have a need to perform traffic scrubbing or cleaning when getting attacked by DDoS volume attacks. They have much larger pipes to the Internet than their internal backbone can actually handle.

Usually a specific IP address is attacked. When that happens, the IP address is placed into MPLS-VRF alternative routing tables, so the traffic gets routed through some scrubbing servers.

The purpose of the scrubbing servers is to reduce (or drop) enough traffic, such that the DoS volume attack is less than the capacity of the internal backbone.

Forward clean traffic

The clean/good traffic needs to be forwarded towards the internal backbone.

To get around the XDP limitation of only sending back out the same NIC, they want to add a VLAN header to the packet before calling *XDP_TX*, allowing them to catch the traffic and re-steer it back into the main MPLS-VRF routing table.

Need: traffic sampling XDP_DROP

They want a way to analyze the traffic they drop (XDP_DROP), to catch false positives. This could be implemented by sampling the drop traffic, by returning XDP_PASS a percentage of the times, and then have a userspace tcpdump running.

To indicate which eBPF rule caused the drop, they were thinking of modifying the packet header by adding a VLAN id. That way the tcpdump could run on a net_device with a given VLAN.

Note: **NEW-ACTION**: The sampling could be implemented more efficiently, if there were a XDP_DUMP action which sent the sampled packets to an AF_PACKET socket.

Need: traffic sampling XDP_TX

If the scrubber filter is not good enough, then too much bad traffic is allowed through. This is usually the base case, once the attack starts.

Thus, they have need for analysing the traffic that gets forwarded with *XDP_TX*. (ISSUE) There is currently no way to sample or dump the *XDP_TX* traffic.

A physical solution could be to do switch-port mirroring of the traffic, and then have another machine (or even the same machine) receive traffic for analysis. They were talking about just using the same machine (as there usually are two NIC ports), but the worry is that this would cost double the PCIe bandwidth.

Warning: **NEW-FEATURE:** A software solution could be a combination of *XDP_TX* and XDP_DUMP. Doing both XDP_TX and XDP_DUMP would only cost an extra page refcnt. They only need sampling. The XDP_DUMP should be implemented such that it has a limited queue size, and simply drops if the queue is full.

Need: smaller eBPF programs

They experience different DDoS attacks. They don't want to have one big eBPF program that needs to handle every kind of attack. This program would also get too slow once the size increase.

DDoS attacks are usually very specific, and are often stopped by spotting a very specific pattern in the packet that is constant enough to identify the bad traffic. It is key that they can quickly construct an XDP program matching this very specific pattern, without risking affecting the stability of other XDP filters.

They also have a need to handle several simultaneous attacks, usually targeting different destination IP addresses.

Warning: **NEED-RXQ-FEATURE**: This could be solved by using NIC HW filters to steer the traffic a specific RX queue, and then allow XDP/eBPF programs to run on specific queues.

Ethtool filters for mlx4

The HW filter capabilities are highly dependent on the HW, and limited by what can be expressed by ethtool.

From below documentation, it looks like mlx4 have the filters needed for this project.

Taken from mlx4 Linux User Manual

Ethtool domain is used to attach an RX ring, specifically its QP to a specified flow. Please refer to the most recent ethtool manpage for all the ways to specify a flow.

Examples:

- ethtool -U mlx4p1 flow-type ether dst f4:52:14:7a:58:f1 loc 5 action 2

  All packets that contain the above destination MAC address are to be steered into rx-ring 2 (its underlying QP), with location/priority 5 (within the ethtool domain)

- ethtool -U mlx4p1 flow-type tcp4 dst-port 22 loc 255 action 2

  All packets that contain the above destination IP address and source port are to be steered into rx-ring 2. When destination MAC is not given, the user's destination MAC is filled automatically.

- ethtool -u mlx4p1

  Shows all of ethtool's steering rule

When configuring two rules with the same location/priority, the second rule will overwrite the first one, so this ethtool interface is effectively a table.

Inserting Flow Steering rules in the kernel requires support from both the ethtool in the user space and in kernel (v2.6.28).

Use-case: Load Balancer

The load-balancer use-case originated from Facebook, as they have a need to load-balance their traffic. They obviously already load balance, but are looking for a faster and more scalable approach.

Facebook currently use the IPVS (IP Virtual Server) load balancer software, which is part of the standard Linux Kernel (since kernel 2.6.10). They even wrote a Python module for configuring IPVS, which is a pure-python replacement for ipvsadm (ipvsadm git tree).

Facebook presented at NetDevConf 2.1 (April 2017) that they are starting to deploy an XDP based solution for both this Load Balancer solution (that gave a 10x speedup compared to IPVS) and a DDoS protection solution named `droplet`. See: slides and YouTube video.

Traditional load balancer

Traditionally a service load balancer (like IPVS) has more NICs (Network Interface Cards), and forwards traffic to the back-end servers (called "real server" for IPVS).

The current XDP implementation (*XDP_TX* in kernel 4.8) can only forward packets back out the same NIC they arrived on. This makes XDP unsuited for implementing a traditional multi-NIC load balancer.

A traditional load balancer easily becomes a single point of failure. Thus, multiple load balancers are usually deployed, in a High Availability (HA) cluster. In order to make load balancer failover transparent to client applications, the load balancer(s) need to synchronize their state (E.g. via IPVS sync protocol sending UDP multicast, preferable send on a separate network/NIC).

Untraditional XDP load balancer

Imagine implementing a load balancer without any dedicated servers for load balancing, 100% scalable and with no single point of failure.

> Be the load balancer yourself!

The main idea is, allow XDP to *be* the load-balancing layer. Running the XDP load balancer software directly on the "back-end" server, with no dedicated central server.

It corresponds to running IPVS on the backend ("real servers"), which is possible, and some IPVS examples are available (e.g Ultra Monkey). But that is generally not recommended (in high load situations), because it increases the load on the application server itself, which leaves less CPU time for serving requests.

> Why is this a good idea for XDP then?

XDP has a speed advantage. The XDP load balance forwarding decision happens **very** early, before the OS has spent/invested too many cycles on the packet. This means the XDP load balancing functionality should not increase the load on the server significantly. Thus, it should be okay to run the service and LB on the same server. One can even imagine having a feedback loop into the LB-program decision, based on whether the service is struggling to keep up.

Who will balance the incoming traffic?

The router can distribute/spread incoming packets across the servers in the cluster, e.g. via using Equal-Cost Multi-Path routing (ECMP) like Google's Maglev solutions does. Google then use some consistent hashing techniques to forward packets to the correct service backend servers.

Serving correct client

The challenging part, in such a distributed system of load balancers, is to coordinate packets getting forwarded to the (correct) server responsible for serving the client.

Google uses a consistent hashing scheme, but other solutions are also possible.

Hardware setup

As mentioned under *Disclaimer*, it is very important to understand hardware environment this kind of setup works within.

When using the same network segment for the load balancing traffic (due to *XDP_TX* limitations), extra care need to be taken when dimensioning the network capacity.

One can create a cluster of servers, all connected to the same 10Gbit/s switch, and the switch has the same 10Gbit/s uplink capacity limitation. The 10Gbit/s capacity is bidirectional, meaning both RX and TX have 10Gbit/s. No (incoming) network overload situation can occur, because the uplink can only forward with 10G, and LB server can RX with 10G and TX with 10G to another "service-server", happening over the Ethernet switch fabric, thus RX capacity of the "service-server" is still 10G. Sending traffic back to the uplink happens via "direct-return" from the "service-server", still have 10G capacity left in the Ethernet switch fabric. Thus, with a proper HW setup the *XDP_TX* limitation can be dealt with.

Need: RX HW hash

Warning: **FEATURE**: provide NIC RX HW hash has as meta-data input to XDP program.

A scheme to determine which **flows** a given server is responsible for serving can benefit from getting the NIC RX hardware hash as input.

The XDP load balancing decision can be made faster, if it does not have to read+parse the packet contents before making a route decision. This is possible if basing the decision on the RX hardware hash, available via the RX descriptor.

Note: Requires: setting up the same NIC HW hash on all servers in the cluster.

End-user documentation

This part of the XDP documentation is targeted at end-users, describing how to use and setup XDP.

The XDP program running (inside the driver hook point) is an eBPF program. eBPF is a general kernel facility not restricted to the XDP use-case. Thus, have its own documentation here: *eBPF - extended Berkeley Packet Filter*. This documentation is focused on using eBPF for the XDP specific use-case.

Contents:

XDP/eBPF build environment

Tool chain

The XDP program running (in the driver hook point) is an eBPF program (see *eBPF - extended Berkeley Packet Filter*). Unless you want to write eBPF machine-code like instruction by hand, you likely want to install some front-ends, that allow you to write some restricted-C code.

Tools for compiling kernel bpf samples requires having installed:

- clang >= version 3.4.0
- llvm >= version 3.7.1

Note that LLVM's tool 'llc' must support target 'bpf', list version and supported targets with command: `llc --version`.

There is also toolkit called BCC (BPF Compiler Collection) that makes eBPF programs easier to write, and front-ends in Python and lua. But it also depend on LLVM.

Build samples/bpf

This github repository also contains some bpf and XDP examples in the directory samples/bpf/. Simply run `make` in that directory to build the bpf samples.

Linux distros

Fedora 25

Since Fedora 25, the package BCC is included with the distribution, and LLVM+clang in the correct versions.

Install commands for Fedora 25:

```
dnf install llvm llvm-libs llvm-doc clang clang-libs
dnf install bcc bcc-tools bcc-doc --enablerepo=updates-testing
dnf install kernel-devel
dnf install python3-pyroute2
```

Note: As of this writing (2017-01-18) BCC for F25 is still in the updates-testing repository.

XDP programs with eBPF

Two projects with example code:

- Using kernel samples/bpf XDP programs and libbpf
- Using BCC toolkit

Kernel samples/bpf

The kernel include some examples of XDP-eBPF programs, see kernel samples/bpf.

There are also some XDP eBPF code examples in the prototype-kernel project under prototype-kernel/kernel/samples/bpf. Simply run `make` inside this directory to compile the samples.

Special XDP eBPF cases

With XDP the eBPF program gets "direct" access to the raw/unstructured packet-data. Thus, eBPF uses some "direct access" instruction for accessing this data, but for safety this need to pass the in-kernel validator.

Walking the packet data, requires writing the boundary checks in a specialized manor.

Like:

```
if (data + nh_off > data_end)
                return rc;
```

CHAPTER 4

Linux Memory Management Subsystem

This is the top-level documentation for the Linux Kernel's Memory Management subsystem.

Contents:

The page_pool documentation

This is top-level for the page_pool documentation.

Contents:

Introduction

The page_pool is a generic API for drivers that have a need for a pool of recycling pages used for streaming DMA.

Motivation

The page_pool is primarily motivated by two things (1) performance and (2) changing the memory model for drivers.

Drivers have developed performance workarounds when the speed of the page allocator and the DMA APIs became too slow for their HW needs. The page pool solves them on a general level providing performance gains and benefits that local driver recycling hacks cannot realize.

A fundamental property is that pages are returned to the page_pool. This property allow a certain class of *Optimization principle*.

Memory model

Once drivers are converted to using page_pool API, then it will become easier to change the underlying memory model backing the driver with pages (without changing the driver).

One prime use-case is NIC zero-copy RX into userspace. As DaveM describes in his Google-plus post, the mapping and unmapping operations in the address space of the process has a cost that cancels out most of the gains of such zero-copy schemes.

This mapping cost can be solved the same way as the keeping DMA mapped trick. By keeping the pages VM-mapped to userspace. This is a layer that can be added later to the page_pool. It will likely be beneficial to also consider using huge-pages (as backing) to reduce the TLB-stress.

Advantages

Advantages of a recycling page pool as bullet points:

1. Faster than going through page-allocator. Given a specialized allocator require less checks, and can piggyback on driver's resource protection (for alloc-side).
2. DMA IOMMU mapping cost is removed by keeping pages mapped.
3. Makes DMA pages writable by predictable DMA unmap point. (**UPDATE** kernel v4.10: This can also be acheived via Alexander Duyck's changes to the DMA API, namely using DMA_ATTR_SKIP_CPU_SYNC, which skips DMA sync as a part the unmap, but requires driver to carefully DMA sync needed memory)
4. OOM protection at device level, as having a feedback-loop knows number of outstanding pages.
5. Flexible memory model allowing zero-copy RX, solving memory early demux (does depend on HW filters into RX queues)
6. Less fragmentation of the page buddy algorithm, when driver maintains a steady-state working-set.

Design: page_pool

Design documentation for the page_pool.

Overall design

The page_pool is designed for performance, and for creating a flexible and common memory model for drivers. Most drivers are based on allocating pages for their DMA receive-rings. Thus, it is a design goal to make it easy to convert these drivers.

Using page_pool provides an immediate performance improvement, and opens up for the longer term goal of zero-copy receive into userspace.

Optimization principle

A fundamental property is that pages **must** be recycled back into the page_pool (when the last user of the page is done).

Recycling pages allow a certain class of optimizations, which is to move setup and tear-down operations out of the fast-path, sometimes known as constructor/destruction operations. DMA map/unmap is one example of operations this applies to. Certain page alloc/free validations can also be avoided in the fast-path. Another example could be pre-mapping pages into userspace, and clearing them (memset-zero) outside the fast-path.

Memory Model

The page_pool should be as transparent as possible. This mean that page coming out of a page_pool, should be considered a normal page (with as few restrictions as possible). This implies a more tight integration with the existing page allocator APIs. (This should also make it easier to compile out.)

Driver are still allowed to split-up page and manipulate refcnt.

DMA map+unmap

The page_pool API takes over the DMA map+unmap operations, based on the *Optimization principle*. The cost of DMA map+unmap depend on the hardware architecture, and whether features like DMA IOMMU have been enabled or not. Thus, the benefit is harder to quantify.

Taking over DMA map+unmap operations, also implies the page_pool cannot be a complete drop-in replacement for the page allocator.

Common driver layer

It is important to have a common layer drivers use for allocating and freeing pages.

The time budget for XDP direct forwarding between interfaces (based on different drivers) cannot rely on pages going through the page allocator (as the base cost is higher than the budget). The page_pool recycle technique is needed here, across drivers.

Drivers also need a flexible memory model for supporting different use-cases, which have trade-offs for different usage scenarios. And the page_pool need to support these scenarios.

Network scenarios: XDP requires drivers to change the memory model to one packet per page. When no XDP program is loaded, the driver can instead choose to conserve memory by splitting up the page to share is for multiple RX packets. When mapping pages to userspace, one packet per page is likely also needed. For more details on networking see *Memory Model for Networking*.

Drivers old memory model

Drivers (not using the page_pool) allocate pages for DMA operations directly from the page allocator. Pages are freed into the page allocator once their refcnt reach zero. Thus, pages are cycles through the page allocator. This actually comes at a fairly high cost, measurable by the page_bench micro-benchmarks and graphs in MM-summit2016 presentation.

Driver work-arounds

> **Warning:** Document not complete

Allocation side

Piggyback on drivers RX protection for page allocations.

Memory Model for Networking

This design describes how the page_pool change the memory model for networking in the NIC (Network Interface Card) drivers.

Note: The catch for driver developers is that, once an application request zero-copy RX, then the driver must use a specific SKB allocation mode and might have to reconfigure the RX-ring.

Design target

Allow the NIC to function as a normal Linux NIC and be shared in a safe manor, between the kernel network stack and an accelerated userspace application using RX zero-copy delivery.

Target is to provide the basis for building RX zero-copy solutions in a memory safe manor. An efficient communication channel for userspace delivery is out of scope for this document, but OOM considerations are discussed below (*Userspace delivery and OOM*).

Background

The SKB or `struct sk_buff` is the fundamental meta-data structure for network packets in the Linux Kernel network stack. It is a fairly complex object and can be constructed in several ways.

From a memory perspective there are two ways depending on RX-buffer/page state:

1. Writable packet page
2. Read-only packet page

To take full potential of the page_pool, the drivers must actually support handling both options depending on the configuration state of the page_pool.

Writable packet page

When the RX packet page is writable, the SKB setup is fairly straight forward. The SKB->data (and skb->head) can point directly to the page data, adjusting the offset according to drivers headroom (for adding headers) and setting the length according to the DMA descriptor info.

The page/data need to be writable, because the network stack need to adjust headers (like TimeToLive and checksum) or even add or remove headers for encapsulation purposes.

A subtle catch, which also requires a writable page, is that the SKB also have an accompanying "shared info" data-structure `struct skb_shared_info`. This "skb_shared_info" is written into the skb->data memory area at the end (skb->end) of the (header) data. The skb_shared_info contains semi-sensitive information, like kernel memory pointers to other pages (which might be pointers to more packet data). This would be bad from a zero-copy point of view to leak this kind of information.

Read-only packet page

When the RX packet page is read-only, the construction of the SKB is significantly more complicated and even involves one more memory allocation.

1. Allocate a new separate writable memory area, and point skb->data here. This is needed due to (above described) skb_shared_info.
2. Memcpy packet headers into this (skb->data) area.
3. Clear part of skb_shared_info struct in writable-area.
4. Setup pointer to packet-data in the page (in skb_shared_info->frags) and adjust the page_offset to be past the headers just copied.

It is useful (later) that the network stack have this notion that part of the packet and a page can be read-only. This implies that the kernel will not "pollute" this memory with any sensitive information. This is good from a zero-copy point of view, but bad from a performance perspective.

NIC RX Zero-Copy

Doing NIC RX zero-copy involves mapping RX pages into userspace. This involves costly mapping and unmapping operations in the address space of the userspace process. Plus for doing this safely, the page memory need to be cleared before using it, to avoid leaking kernel information to userspace, also a costly operation. The page_pool base "class" of optimization is moving these kind of operations out of the fastpath, by recycling and lifetime control.

Once a NIC RX-queue's page_pool have been configured for zero-copy into userspace, then can packets still be allowed to travel the normal stack?

Yes, this should be possible, because the driver can use the SKB-read-only mode, which avoids polluting the page data with kernel-side sensitive data. This implies, when a driver RX-queue switch page_pool to RX-zero-copy mode it MUST also switch to SKB-read-only mode (for normal stack delivery for this RXq).

XDP can be used for controlling which pages that gets RX zero-copied to userspace. The page is still writable for the XDP program, but read-only for normal stack delivery.

Kernel safety

For the paranoid, how do we protect the kernel from a malicious userspace program. Sure there will be a communication interface between kernel and userspace, that synchronize ownership of pages. But a userspace program can violate this interface, given pages are kept VMA mapped, the program can in principle access all the memory pages in the given page_pool. This opens up for a malicious (or defect) program modifying memory pages concurrently with the kernel and DMA engine using them.

An easy way to get around userspace modifying page data contents is simply to map pages read-only into userspace.

Note: The first implementation target is read-only zero-copy RX page to userspace and require driver to use SKB-read-only mode.

Advanced: Allowing userspace write access?

What if userspace need write access? Flipping the page permissions per transfer will likely kill performance (as this likely affects the TLB-cache).

I will argue that giving userspace write access is still possible, without risking a kernel crash. This is related to the SKB-read-only mode that copies the packet headers (in to another memory area, inaccessible to userspace). The attack angle is to modify packet headers after they passed some kernel network stack validation step (as once headers are copied they are out of "reach").

Situation classes where memory page can be modified concurrently:

1. When DMA engine owns the page. Not a problem, as DMA engine will simply overwrite data.
2. Just after DMA engine finish writing. Not a problem, the packet will go through netstack validation and be rejected.
3. While XDP reads data. This can lead to XDP/eBPF program goes into a wrong code branch, but the eBPF virtual machine should not be able to crash the kernel. The worst outcome is a wrong or invalid XDP return code.
4. Before SKB with read-only page is constructed. Not a problem, the packet will go through netstack validation and be rejected.
5. After SKB with read-only page has been constructed. Remember the packet headers were copied into a separate memory area, and the page data is pointed to with an offset passed the copied headers. Thus, userspace cannot modify the headers used for netstack validation. It can only modify packet data contents, which is less critical as it cannot crash the kernel, and eventually this will be caught by packet checksum validation.
6. After netstack delivered packet to another userspace process. Not a problem, as it cannot crash the kernel. It might corrupt packet-data being read by another userspace process, which one argument for requiring elevated privileges to get write access (like NET_CAP_ADMIN).

Userspace delivery and OOM

These RX pages are likely mapped to userspace via mmap(), so-far so good. It is key to performance to get an efficient way of signaling between kernel and userspace, e.g what page are ready for consumption, and when userspace are done with the page.

It is outside the scope of page_pool to provide such a queuing structure, but the page_pool can offer some means of protecting the system resource usage. It is a classical problem that resources (e.g. the page) must be returned in a timely manor, else the system, in this case, will run out of memory. Any system/design with unbounded memory allocation can lead to Out-Of-Memory (OOM) situations.

Communication between kernel and userspace is likely going to be some kind of queue. Given transferring packets individually will have too much scheduling overhead. A queue can implicitly function as a bulking interface, and offers a natural way to split the workload across CPU cores.

This essentially boils down-to a two queue system, with the RX-ring queue and the userspace delivery queue.

Two bad situations exists for the userspace queue:

1. Userspace is not consuming objects fast-enough. This should simply result in packets getting dropped when enqueueing to a full userspace queue (as queue *must* implement some limit). Open question is; should this be reported or communicated to userspace.
2. Userspace is consuming objects fast, but not returning them in a timely manor. This is a bad situation, because it threatens the system stability as it can lead to OOM.

The page_pool should somehow protect the system in case 2. The page_pool can detect the situation as it is able to track the number of outstanding pages, due to the recycle feedback loop. Thus, the page_pool can have some configurable limit of allowed outstanding pages, which can protect the system against OOM.

Note, the Fbufs paper propose to solve case 2 by allowing these pages to be "pageable", i.e. swap-able, but that is not an option for the page_pool as these pages are DMA mapped.

Effect of blocking allocation

The effect of page_pool, in case 2, that denies more allocations essentially result-in the RX-ring queue cannot be refilled and HW starts dropping packets due to "out-of-buffers". For NICs with several HW RX-queues, this can be limited to a subset of queues (and admin can control which RX queue with HW filters).

The question is if the page_pool can do something smarter in this case, to signal the consumers of these pages, before the maximum limit is hit (of allowed outstanding packets). The MM-subsystem already have a concept of emergency PFMEMALLOC reserves and associate page-flags (e.g. page_is_pfmemalloc). And the network stack already handle and react to this. Could the same PFMEMALLOC system be used for marking pages when limit is close?

This requires further analysis. One can imagine; this could be used at RX by XDP to mitigate the situation by dropping less-important frames. Given XDP choose which pages are being send to userspace it might have appropriate knowledge of what it relevant to drop(?).

Note: An alternative idea is using a data-structure that blocks userspace from getting new pages before returning some. (out of scope for the page_pool)

Early demux problem

Todo

Describe the early demux problem, and how page_pool solves this.

CHAPTER 5

eBPF - extended Berkeley Packet Filter

Introduction

The Berkeley Packet Filter (BPF) started (article 1992) as a special-purpose virtual machine (register based filter evaluator) for filtering network packets, best known for its use in tcpdump. It is documented in the kernel tree, in the first part of: Documentation/networking/filter.txt

The extended BPF (eBPF) variant has become a universal in-kernel virtual machine, that has hooks all over the kernel. The eBPF instruction set is quite different, see description in section "BPF kernel internals" of Documentation/networking/filter.txt or look at this presentation by Alexei.

Areas using eBPF:

- *XDP - eXpress Data Path*
- Traffic control
- Sockets
- Firewalling (`xt_bpf` module)
- Tracing
- Tracepoints
- kprobe (dynamic tracing of a kernel function call)
- cgroups

Documentation

The primary user documentation for extended BPF is in the man-page for the bpf(2) syscall.

An excellent BPF and XDP Reference Guide is being maintained by the Cilium project.

This documentation is focused on the kernel tree's samples/bpf/ and tools/lib/bpf/. It is worth mentioning that other projects exist, like *BCC (BPF Compiler Collection)*, that has a slightly different user-facing syntax, but is interfacing with the same kernel facilities as those covered by this documentation.

eBPF maps

This document describes what eBPF maps are, how you create them (*Creating a map*), and how to interact with them (*Interacting with maps*). The different map types available are described here: *Types of eBPF maps*.

Using eBPF maps is a method to keep state between invocations of the eBPF program, and allows sharing data between eBPF kernel programs, and also between kernel and user-space applications.

Basically a key/value store with arbitrary structure (from man-page bpf(2)):

> eBPF maps are a generic data structure for storage of different data types. Data types are generally treated as binary blobs, so a user just specifies the size of the key and the size of the value at map-creation time. In other words, a key/value for a given map can have an arbitrary structure.

The map handles are file descriptors, and multiple maps can be created and accessed by multiple programs (from man-page bpf(2)):

> A user process can create multiple maps (with key/value-pairs being opaque bytes of data) and access them via file descriptors. Different eBPF programs can access the same maps in parallel. It's up to the user process and eBPF program to decide what they store inside maps.

Creating a map

A map is created based on a request from userspace, via the bpf syscall (specifically bpf_cmd BPF_MAP_CREATE), which returns a new file descriptor that refers to the map. On error, -1 is returned and errno is set to EINVAL, EPERM, or ENOMEM. These are the struct `bpf_attr` setup arguments to use when creating a map via the syscall:

```
bpf(BPF_MAP_CREATE, &bpf_attr, sizeof(bpf_attr));
```

Notice how this kernel ABI is extensible, as more struct arguments can easily be added later as the sizeof(bpf_attr) is passed along to the syscall. This also implies that API users must clear/zero sizeof(bpf_attr), as compiler can size-align the struct differently, to avoid garbage data to be interpreted as parameters by future kernels.

The following configuration attributes are needed when creating the map:

```
union bpf_attr {
 struct { /* anonymous struct used by BPF_MAP_CREATE command */
        __u32   map_type;       /* one of enum bpf_map_type */
        __u32   key_size;       /* size of key in bytes */
        __u32   value_size;     /* size of value in bytes */
        __u32   max_entries;    /* max number of entries in a map */
        __u32   map_flags;      /* prealloc or not */
 };
}
```

Kernel sample/bpf ELF convention

For programs under samples/bpf/, defining a map have been integrated with ELF binary generated by LLVM. This is purely one example of a userspace convention and not part of the kernel ABI. It still invokes the bpf syscall.

Map definitions are done by defining a `struct bpf_map_def` with an elf section __attribute__ `SEC("maps")`, in the xxx_kern.c file. The maps file descriptor is available in the userspace xxx_user.c file, via global array variable

`map_fd[]`, and the array map index corresponds to the order the maps sections were defined in elf file of xxx_kern.c file. Behind the scenes it is the `load_bpf_file()` call (from samples/bpf/bpf_load) that takes care of parsing ELF file compiled by LLVM, pickup 'maps' section and creates maps via the bpf syscall.

```
struct bpf_map_def {
      unsigned int type;
      unsigned int key_size;
      unsigned int value_size;
      unsigned int max_entries;
      unsigned int map_flags;
};

struct bpf_map_def SEC("maps") my_map = {
      .type        = BPF_MAP_TYPE_XXX,
      .key_size    = sizeof(u32),
      .value_size  = sizeof(u64),
      .max_entries = 42,
      .map_flags   = 0
};
```

Qdisc Traffic Control convention

It is worth mentioning, that qdisc TC (Traffic Control), also use ELF files for defining the maps, but it uses another layout. See man-page tc-bpf(8) and tc bpf examples in iproute2.git tree.

Interacting with maps

Interacting with eBPF maps happens through some **lookup/update/delete** primitives.

When writing eBFP programs using load helpers and libraries from samples/bpf/ and tools/lib/bpf/. Common function name API have been created that hides the details of how kernel vs. userspace access these primitives (which is quite different).

The common function names (parameters and return values differs):

```
void bpf_map_lookup_elem(map, void *key. ...);
void bpf_map_update_elem(map, void *key, ..., __u64 flags);
void bpf_map_delete_elem(map, void *key);
```

The `flags` argument in `bpf_map_update_elem()` allows to define semantics on whether the element exists:

```
/* File: include/uapi/linux/bpf.h */
/* flags for BPF_MAP_UPDATE_ELEM command */
#define BPF_ANY       0 /* create new element or update existing */
#define BPF_NOEXIST   1 /* create new element only if it didn't exist */
#define BPF_EXIST     2 /* only update existing element */
```

Userspace

The userspace API map helpers are defined in tools/lib/bpf/bpf.h and looks like this:

```
/* Userspace helpers */
int bpf_map_lookup_elem(int fd, void *key, void *value);
int bpf_map_update_elem(int fd, void *key, void *value, __u64 flags);
```

```
int bpf_map_delete_elem(int fd, void *key);
/* Only userspace: */
int bpf_map_get_next_key(int fd, void *key, void *next_key);
```

Interacting with an eBPF map from **userspace**, happens through the bpf syscall and a file descriptor. See how the map handle `int fd` is a file descriptor . On success, zero is returned, on failures -1 is returned and errno is set.

Wrappers for the bpf syscall is implemented in tools/lib/bpf/bpf.c, and ends up calling functions in kernel/bpf/syscall.c, like map_lookup_elem.

```
/* Corresponding syscall bpf commands from userspace */
enum bpf_cmd {
      [...]
      BPF_MAP_LOOKUP_ELEM,
      BPF_MAP_UPDATE_ELEM,
      BPF_MAP_DELETE_ELEM,
      BPF_MAP_GET_NEXT_KEY,
      [...]
};
```

Notice how `void *key` and `void *value` are passed as a void pointers. Given the memory seperation between kernel and userspace, this is a copy of the value. Kernel primitives like `copy_from_user()` and `copy_to_user()` are used, e.g. see map_lookup_elem, which also kmalloc+kfree memory for a short period.

From userspace, there is no function call to atomically increment or decrement the value 'in-place'. The bpf_map_update_elem() call will overwrite the existing value, with a copy of the value supplied. Depending on the map type, the overwrite will happen in an atomic way, e.g. using locking mechanisms specific to the map type.

Kernel-side eBPF program

The API mapping for eBPF programs on the kernel-side is fairly hard to follow. It related to samples/bpf/bpf_helpers.h and maps into kernel/bpf/helpers.c via macros.

```
/* eBPF program helpers */
void *bpf_map_lookup_elem(void *map, void *key);
int bpf_map_update_elem(void *map, void *key, void *value, unsigned long long flags);
int bpf_map_delete_elem(void *map, void *key);
```

The eBPF-program running kernel-side interacts more directly with the map data structures. For example the call `bpf_map_lookup_elem()` returns a direct pointer to the 'value' memory-element inside the kernel (while userspace gets a copy). This allows the eBPF-program to atomically increment or decrement the value 'in-place', by using appropiate compiler primitives like `__sync_fetch_and_add()`, which is understood by LLVM when generating eBPF instructions.

Todo

1. describe how verifier validate map access to be safe.
2. describe int return codes of bpf_map_update_elem + bpf_map_delete_elem.

Export map to filesystem

When *Interacting with maps* from *Userspace* a file descriptor is needed. There are two methods for sharing this file descriptor.

1. By passing it over Unix-domain sockets.
2. Exporting the map to a special `bpf` filesystem.

Option 2, exporting or pinning the map through the filesystem is more convenient and easier than option 1. Thus, this document will focus on option 2.

Todo

Describe the API for `bpf_obj_pin` and `bpf_obj_get`. Usage examples available in XDP blacklist for bpf_obj_pin() and XDP blacklist cmdline tool show use of bpf_obj_get().

Todo

add link to Daniel's TC example of using Unix-domain sockets.

Types of eBPF maps

This document describes the different types of eBPF maps available, and goes into details about the individual map types. The purpose is to help choose the right type based on the individual use-case. Creating and interacting with maps are described in another document here: *eBPF maps*.

The different types of maps available, are defined by `enum bpf_map_type` in include/uapi/linux/bpf.h. These type definition "names" are needed when creating the map. Example of `bpf_map_type`, but remember to lookup latest available maps in the source code.

```
enum bpf_map_type {
        BPF_MAP_TYPE_UNSPEC,
        BPF_MAP_TYPE_HASH,
        BPF_MAP_TYPE_ARRAY,
        BPF_MAP_TYPE_PROG_ARRAY,
        BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        BPF_MAP_TYPE_PERCPU_HASH,
        BPF_MAP_TYPE_PERCPU_ARRAY,
        BPF_MAP_TYPE_STACK_TRACE,
        BPF_MAP_TYPE_CGROUP_ARRAY,
        BPF_MAP_TYPE_LRU_HASH,
        BPF_MAP_TYPE_LRU_PERCPU_HASH,
};
```

Implementation details

In-order to understand and follow the descriptions of the different map types, in is useful for the reader to understand how a map type is implemented by the kernel.

On the kernel side, implementing a map type requires defining some function call (pointers) via struct bpf_map_ops. The eBPF programs (and userspace) have access to the functions calls `map_lookup_elem`, `map_update_elem` and `map_delete_elem`, which get invoked from eBPF via bpf-helpers in kernel/bpf/helpers.c, or via userspace the bpf syscall (as described in *eBPF maps*).

Creating a map requires supplying the following configuration attributes: map_type, key_size, value_size, max_entries and map_flags.

BPF_MAP_TYPE_ARRAY

Implementation defined in kernel/bpf/arraymap.c via struct bpf_map_ops array_ops.

As the name `BPF_MAP_TYPE_ARRAY` indicates, this can be seen as an array. All array elements are pre-allocated and zero initialized at init time. Key is an index in array and can only be 4 bytes (32-bit). The constant size is defined by `max_entries`. This init-time constant also implies bpf_map_delete_elem (array_map_delete_elem) is an invalid operation.

Optimized for fastest possible lookup. The size is constant for the life of the eBPF program, which allows verifier+JIT to perform a wider range of optimizations. E.g. array_map_lookup_elem() may be 'inlined' by JIT.

Small size gotcha, the `value_size` is rounded up to 8 bytes.

Example usage BPF_MAP_TYPE_ARRAY, based on samples/bpf/sockex1_kern.c:

```
struct bpf_map_def SEC("maps") my_map = {
      .type = BPF_MAP_TYPE_ARRAY,
      .key_size = sizeof(u32),
      .value_size = sizeof(long),
      .max_entries = 256,
};

u32 index = 42;
long *value;
value = bpf_map_lookup_elem(&my_map, &index);
      if (value)
              __sync_fetch_and_add(value, 1);
```

The lookup (from kernel side) `bpf_map_lookup_elem()` returns a pointer into the array element. To avoid data races with userspace reading the value, the API-user must use primitives like `__sync_fetch_and_add()` when updating the value in-place.

Troubleshooting eBPF

This document should help end-users with troubleshooting their eBPF programs. With a primary focus on programs under kernels samples/bpf.

Memory ulimits

The eBPF maps uses locked memory, which is default very low. Your program likely need to increase resource limit `RLIMIT_MEMLOCK` see system call setrlimit(2).

The `bpf_create_map` call will return errno EPERM (Operation not permitted) when the RLIMIT_MEMLOCK memory size limit is exceeded.

Enable bpf JIT

Not seeing the expected performance and perf top showing `__bpf_prog_run()` as the top CPU consumer.

Did you remember to enable JIT'ing of the BPF code? Like:

```
$ sysctl net/core/bpf_jit_enable=1
net.core.bpf_jit_enable = 1
```

Notice there is both JIT'ing of eBPF and cBPF (Classical BPF) implemented in the kernel per arch. You can see current cBPF and eBPF JITs that are supported by the kernel via:

```
$ git grep BPF_JIT | grep select
arch/arm/Kconfig:      select HAVE_CBPF_JIT
arch/arm64/Kconfig:    select HAVE_EBPF_JIT
arch/mips/Kconfig:     select HAVE_CBPF_JIT if !CPU_MICROMIPS
arch/powerpc/Kconfig:  select HAVE_CBPF_JIT                    if !PPC64
arch/powerpc/Kconfig:  select HAVE_EBPF_JIT                    if PPC64
arch/s390/Kconfig:     select HAVE_EBPF_JIT if PACK_STACK && HAVE_MARCH_Z196_FEATURES
arch/sparc/Kconfig:    select HAVE_CBPF_JIT if SPARC32
arch/sparc/Kconfig:    select HAVE_EBPF_JIT if SPARC64
arch/x86/Kconfig:      select HAVE_EBPF_JIT                    if X86_64
```

Also see Cilium JIT section and BPF sysctl section.

ELF binary

The binary containing the eBPF program, which got generated by the LLVM compiler, is an normal ELF binary. For samples/bpf/ this is the file named xxx_kern.o. It is possible to inspect this normal ELF file, with tools like `readelf` or `llvm-objdump`.

```
$ llvm-objdump -h xdp_ddos01_blacklist_kern.o

xdp_ddos01_blacklist_kern.o:   file format ELF64-unknown

Sections:
Idx Name          Size      Address          Type
 0                00000000 0000000000000000
 1 .strtab        00000072 0000000000000000
 2 .text          00000000 0000000000000000 TEXT DATA
 3 xdp_prog       000001b8 0000000000000000 TEXT DATA
 4 .relxdp_prog   00000020 0000000000000000
 5 maps           00000028 0000000000000000 DATA
 6 license        00000004 0000000000000000 DATA
 7 .symtab        000000d8 0000000000000000
```

From the above output some trivial information can be extracted. This is an XDP program, as the defined program section Idx 3 starts with the letters "xdp". From the same line the size column also show the program size in hex 0001b8 equal 440 bytes, or 55 bpf instructions, as each insns is 8 bytes (see struct bpf_insn) (shell trick `echo $((0x1b8)) insns=$((0x1b8 / 8))`). Do notice this size is not the JIT'ed program size.

The loader code samples/bpf/bpf_load.c parse this elf file, extract needed program sections, uses the maps section and relocation section (here .relxdp_prog) to remap the BPF_PSEUDO_MAP_FD instruction to point to the correct map (which gets created during parsing of the maps section, via standard bpf-syscall bpf_create_map).

LLVM disassemble support

Todo

Document what LLVM version this "-S" option got added

In newer versions of LLVM, the tool `llvm-objdump`, supports showing section names, asm code and original C code, if compiled with `-g`.

```
llvm-objdump -S prog_kern.o
```

Todo

What does the option -no-show-raw-insn do?

See Cilium Toolchain LLVM section for more details.

Extracting eBPF-JIT code

Also see Cilium JIT Debugging.

For debugging/seeing the generated JIT code, is it possible to change this proc sysctl:

```
sysctl net.core.bpf_jit_enable=2
```

The output looks like:

```
flen=55 proglen=335 pass=4 image=ffffffffa0006820 from=xdp_ddos01_blac pid=13333
JIT code: 00000000: 55 48 89 e5 48 81 ec 28 02 00 00 48 89 9d d8 fd
JIT code: 00000010: ff ff 4c 89 ad e0 fd ff ff 4c 89 b5 e8 fd ff ff
JIT code: 00000020: 4c 89 bd f0 fd ff ff 31 c0 48 89 85 f8 fd ff ff
JIT code: 00000030: bb 02 00 00 00 48 8b 77 08 48 8b 7f 00 48 89 fa
JIT code: 00000040: 48 83 c2 0e 48 39 f2 0f 87 e1 00 00 00 48 0f b6
JIT code: 00000050: 4f 0c 48 0f b6 57 0d 48 c1 e2 08 48 09 ca 48 89
JIT code: 00000060: d1 48 81 e1 ff 00 00 00 41 b8 06 00 00 00 49 39
JIT code: 00000070: c8 0f 87 b7 00 00 00 48 81 fa 88 a8 00 00 74 0e
JIT code: 00000080: b9 0e 00 00 00 48 81 fa 81 00 00 00 75 1a 48 89
JIT code: 00000090: fa 48 83 c2 12 48 39 f2 0f 87 90 00 00 00 b9 12
JIT code: 000000a0: 00 00 00 48 0f b7 57 10 bb 02 00 00 00 48 81 e2
JIT code: 000000b0: ff ff 00 00 48 83 fa 08 75 49 48 01 cf 31 db 48
JIT code: 000000c0: 89 fa 48 83 c2 14 48 39 f2 77 38 8b 7f 0c 89 7d
JIT code: 000000d0: fc 48 89 ee 48 83 c6 fc 48 bf 00 9c 24 5f 07 88
JIT code: 000000e0: ff ff e8 29 cd 13 e1 bb 02 00 00 00 48 83 f8 00
JIT code: 000000f0: 74 11 48 8b 78 00 48 83 c7 01 48 89 78 00 bb 01
JIT code: 00000100: 00 00 00 89 5d f8 48 89 ee 48 83 c6 f8 48 bf c0
JIT code: 00000110: 76 12 13 04 88 ff ff e8 f4 cc 13 e1 48 83 f8 00
JIT code: 00000120: 74 0c 48 8b 78 00 48 83 c7 01 48 89 78 00 48 89
JIT code: 00000130: d8 48 8b 9d d8 fd ff ff 4c 8b ad e0 fd ff ff 4c
JIT code: 00000140: 8b b5 e8 fd ff ff 4c 8b bd f0 fd ff ff c9 c3
```

The `proglen` is the len of opcode sequence generated and `flen` is the number of bpf insns. You can use tools/net/bpf_jit_disasm.c to disassemble that output. bpf_jit_disasm -o will dump the related opcodes as well.

Perf tool symbols

For JITed progs, you can do sysctl net/core/bpf_jit_kallsyms=1 and f.e. perf script –kallsyms=/proc/kallsyms to show them based on the tag:

```
sysctl net/core/bpf_jit_kallsyms=1
```

Detail see commit: https://git.kernel.org/torvalds/c/74451e66d516c55e3

Remember to use the perf command-line option –kallsyms=/proc/kallsyms to get the symobols resolved, like:

```
# perf report --no-children --kallsyms=/proc/kallsyms
```

BCC (BPF Compiler Collection)

BCC is a toolkit to make eBPF programs easier to write, with front-ends in Python and Lua. BCC requires LLVM and clang (in version 3.7.1 or newer) to be available on target, because BCC programs do runtime compilation of the restricted-C code into eBPF instructions.

BCC includes several useful tools and examples, developed by recognized performance analyst Brendan Gregg and covered with a tutorial and slides.

The project maintains an overview of eBPF supported kernels and what versions got which specific features. There is also a BCC Reference Guide.

CHAPTER 6

Blogposts, Reports and Write-ups

This documentation area is used for publishing reports and write-ups of my work, which I find relevant for other people. This will closely resemble previous my blogposts. Hoping using this rst-text format will make it less of an hassle to publish my work. (This directory is not intented to be integrated with the kernels documentation tree).

Contents:

Eval Generic netstack XDP patch

Authors Jesper Dangaard Brouer

Version 1.0.1

Date 2017-04-24 Mon

Updated 2017-06-08

Given XDP works at the driver level, developing and testing XDP programs requires access to specific NIC hardware... but this is about to change in kernel v4.12.

> **UPDATE (2017-06-08)**: The mentioned/evaluated patches have been accepted and will appear in kernel release v4.12

To ease developing and testing XDP programs, a generic netstack-XDP patch proposal (PATCH V3 and PATCH V4) have been posted. This allow for attaching XDP programs to any net_device. If the driver doesn't support native XDP, the XDP eBPF program gets attached further inside the network stack. This is obviously slower and loses the XDP benefit of skipping the SKB allocation.

The generic netstack-XDP patchset is **NOT** targetted high performance, but instead for making it easier to test and develop XDP programs.

That said, this does provide an excellent opportunity for comparing performance between NIC-level-XDP and netstack-XDP. This provides the ability to do what I call zoom-in-benchmarking of the network stack facilities, that the NIC-XDP programs avoid. Thus, allowing us to quantify the cost of these facilities.

Special note for the KVM driver `virtio_net`:

XDP support have been added to KVM via the virtio_net driver, but unfortunately it is a hassle to configure (given it requires disabling specific options, which are default enabled).

Benchmark program

The XDP program used is called: `xdp_bench01_mem_access_cost` and is available in the prototype kernel samples/bpf directory as xdp_bench01_mem_access_cost_kern.c and _user.c.

UPDATE (2017-06-08): The `xdp_bench01_mem_access_cost` program have gotten an option called `--skb-mode`, which will force using "Generic XDP" even on interfaces that do support XDP natively. This is practical for doing this kind of comparison as described in the document.

Baseline testing with NIC-level XDP

First establish a baseline for the performance of NIC-level XDP. This will serve as baseline against the patch being evaluated. The packet generator machine is running pktgen_sample03_burst_single_flow.sh, which implies these tests are single CPU RX performance, as the UDP flow will hit a single hardware RX-queue, and thus only activate a single CPU.

Baseline with mlx5 on a Skylake CPU: Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz.

Network card (NIC) hardware: NIC: ConnectX-4 Dual 100Gbit/s, driver mlx5. Machines connected back-to-back with Ethernet-flow control disabled.

Dropping packet without touching the packet data (thus avoiding cache-miss) have a huge effect on my system. HW indicate via PMU counter LLC-load-misses that DDIO is working on my system, but the L3-to-L1 cache-line miss is causing the CPU to stall:

```
[jbrouer@skylake prototype-kernel]$
 sudo ./xdp_bench01_mem_access_cost --action XDP_DROP --dev mlx5p2
XDP_action   pps        pps-human-readable mem
XDP_DROP     19851067   19,851,067         no_touch
XDP_DROP     19803663   19,803,663         no_touch  (**used in examples**)
XDP_DROP     19795927   19,795,927         no_touch
XDP_DROP     19792161   19,792,161         no_touch
XDP_DROP     19792109   19,792,109         no_touch
```

I have previously posted patches to the mlx5 and mlx4 driver, that prefetch packet-data into L2, and avoid this cache stall, and I can basically achieve same result as above, even when reading data. Mellanox have taken over these patches, but they are stalling on that on newer E5-26xx v4 CPUs this prefetch already happens in HW.

This is a more realistic XDP_DROP senario where we touch packet data before dropping it (causes cache miss from L3):

```
[jbrouer@skylake prototype-kernel]$
  sudo ./xdp_bench01_mem_access_cost --action XDP_DROP --dev mlx5p2 --read
XDP_action   pps        pps-human-readable mem
XDP_DROP     11972515   11,972,515         read
XDP_DROP     12006685   12,006,685         read     (**used in examples**)
XDP_DROP     12004640   12,004,640         read
XDP_DROP     11997837   11,997,837         read
XDP_DROP     11998538   11,998,538         read
^CInterrupted: Removing XDP program on ifindex:5 device:mlx5p2
```

An interesting observation and take-ways from these two measurements is that this cache-miss cost approx 32ns ((1/12006685-1/19803663)*10^9).

For the XDP_TX test to be correct, it is important to swap MAC-addrs else the NIC HW will not transmit this to the wire (I verified this was actually TX'ed to the wire):

```
[jbrouer@skylake prototype-kernel]$
 sudo ./xdp_bench01_mem_access_cost --action XDP_TX --dev mlx5p2 --read
XDP_action   pps        pps-human-readable mem
XDP_TX       10078899   10,078,899         read
XDP_TX       10109107   10,109,107         read
XDP_TX       10107393   10,107,393         read
XDP_TX       10107946   10,107,946         read
XDP_TX       10109020   10,109,020         read
```

Testing with network stack generic XDP

This test is based on PATCH V4 after adjusting the patch according to the email thread, and and validated XDP_TX can send packets on wire.

Netstack XDP_DROP

As expected there is no difference in letting the XDP prog touch/read packet-data vs "no_touch", because we cannot avoid touching given the XDP/eBPF hook happens much later in the network stack. As can be seen by these benchmarks:

```
[jbrouer@skylake prototype-kernel]$
 sudo ./xdp_bench01_mem_access_cost --action XDP_DROP --dev mlx5p2
XDP_action   pps        pps-human-readable mem
XDP_DROP     8438488    8,438,488          no_touch
XDP_DROP     8423788    8,423,788          no_touch
XDP_DROP     8425617    8,425,617          no_touch
XDP_DROP     8421396    8,421,396          no_touch
XDP_DROP     8432846    8,432,846          no_touch
^CInterrupted: Removing XDP program on ifindex:7 device:mlx5p2
```

The drop numbers are good, for the netstack but some distance to the 12,006,685 pps of XDP running on in-the-NIC. Percentage-wise it looks big a reduction of approx 30%. But nanosec difference is it "only" (1/12006685*10^9)-(1/8413417*10^9) = -35.57 ns

```
[jbrouer@skylake prototype-kernel]$
 sudo ./xdp_bench01_mem_access_cost --action XDP_DROP --dev mlx5p2 --read
XDP_action   pps        pps-human-readable mem
XDP_DROP     8415835    8,415,835          read
XDP_DROP     8413417    8,413,417          read
XDP_DROP     8236525    8,236,525          read
XDP_DROP     8410996    8,410,996          read
XDP_DROP     8412015    8,412,015          read
^CInterrupted: Removing XDP program on ifindex:7 device:mlx5p2
```

Do notice, that reaching around 8Mpps is a **very** good result for the normal networks stack, because 100Gbit/s with large MTU size frames (1536 bytes due to overheads) corresponds to 8,138,020 pps ((100*10^9)/(1536*8)). The above test is with small 64bytes packets, and the generator sending 40Mpps (can be tuned to 65Mpps).

Below perf-stat for this generic netstack-XDP_DROP test, show a high (2.01) insn per cycle indicate that it is functioning fairly optimal, and we likely cannot find any "magic" trick as the CPU does not seem to be stalling on something:

```
$ sudo ~/perf stat -C7 -e L1-icache-load-misses -e cycles:k \
  -e  instructions:k -e cache-misses:k -e   cache-references:k \
  -e LLC-store-misses:k -e LLC-store -e LLC-load-misses:k \
  -e  LLC-load -r 4 sleep 1

Performance counter stats for 'CPU(s) 7' (4 runs):

      349,830  L1-icache-load-misses                  ( +-  0.53% )  (33.31%)
3,989,134,732  cycles:k                               ( +-  0.06% )  (44.50%)
8,016,054,916  instructions:k # 2.01  insn per cycle       (+- 0.02%) (55.62%)
   31,843,544  cache-misses:k # 17.337 % of all cache refs (+- 0.04%) (66.71%)
  183,671,576  cache-references:k                     ( +-  0.03% )  (66.71%)
    1,190,204  LLC-store-misses                       ( +-  0.29% )  (66.71%)
   17,376,723  LLC-store                              ( +-  0.04% )  (66.69%)
       55,058  LLC-load-misses                        ( +-  0.07% )  (22.19%)
    3,056,972  LLC-load                               ( +-  0.13% )  (22.19%)
```

Netstack XDP_TX

When testing XDP_TX it is important to verify that packets are actually transmitted. This is because the NIC HW can choose to drop invalid packets, which changes the performance profile and your results.

Generic netstack-XDP_TX verified actually hitting wire. The slowdown is higher than expected. Maybe we are stalling on the tairptr/doorbell update on TX???

```
[jbrouer@skylake prototype-kernel]$
 sudo ./xdp_bench01_mem_access_cost --action XDP_TX --dev mlx5p2 --read
XDP_action   pps        pps-human-readable mem
XDP_TX       4577542    4,577,542          read
XDP_TX       4484903    4,484,903          read
XDP_TX       4571821    4,571,821          read
XDP_TX       4574512    4,574,512          read
XDP_TX       4574424    4,574,424          read     (**use in examples**)
XDP_TX       4575712    4,575,712          read
XDP_TX       4505569    4,505,569          read
^CInterrupted: Removing XDP program on ifindex:7 device:mlx5p2
```

Below perf-stat for generic netstack-XDP_TX, show a lower (1.51) insn per cycle, indicate that the system is stalling on something

```
$ sudo ~/perf stat -C7 -e L1-icache-load-misses -e cycles:k \
 -e  instructions:k -e cache-misses:k -e   cache-references:k \
 -e LLC-store-misses:k -e LLC-store -e LLC-load-misses:k \
 -e  LLC-load -r 4 sleep 1

Performance counter stats for 'CPU(s) 7' (4 runs):

      518,261  L1-icache-load-misses        ( +-  0.58% )  (33.30%)
3,989,223,247  cycles:k                     ( +-  0.01% )  (44.49%)
6,017,445,820  instructions:k #  1.51  insn per cycle      (+- 0.31%) (55.62%)
   26,931,778  cache-misses:k # 10.930 % of all cache refs (+- 0.05%) (66.71%)
  246,406,110  cache-references:k           ( +-  0.19% )  (66.71%)
    1,317,850  LLC-store-misses             ( +-  2.93% )  (66.71%)
   30,028,771  LLC-store                    ( +-  0.88% )  (66.70%)
       72,146  LLC-load-misses              ( +-  0.22% )  (22.19%)
   12,426,426  LLC-load                     ( +-  2.12% )  (22.19%)
```

Perf details for netstack XDP_TX

My first though is that there is a high probability that this could be the tairptr/doorbell update. Looking at perf report something else lights up, which could still be the tailptr, as it stalls on the next lock operation

```
Samples: 25K of event 'cycles', Event count (approx.): 25790301710
 Overhead  Symbol
+   24.75%  [k] mlx5e_handle_rx_cqe
+   16.95%  [k] __build_skb
+   10.72%  [k] mlx5e_xmit
+    7.03%  [k] build_skb
+    5.31%  [k] mlx5e_alloc_rx_wqe
+    2.99%  [k] kmem_cache_alloc
+    2.65%  [k] ___slab_alloc
+    2.65%  [k] _raw_spin_lock
+    2.52%  [k] bpf_prog_662b9cae761bf6ab
+    2.37%  [k] netif_receive_skb_internal
+    1.92%  [k] memcpy_erms
+    1.73%  [k] generic_xdp_tx
+    1.69%  [k] mlx5e_get_cqe
+    1.40%  [k] __netdev_pick_tx
+    1.28%  [k] __rcu_read_unlock
+    1.19%  [k] netdev_pick_tx
+    1.02%  [k] swiotlb_map_page
+    1.00%  [k] __cmpxchg_double_slab.isra.56
+    0.99%  [k] dev_gro_receive
+    0.85%  [k] __rcu_read_lock
+    0.80%  [k] napi_gro_receive
+    0.79%  [k] mlx5e_poll_rx_cq
+    0.73%  [k] mlx5e_post_rx_wqes
+    0.71%  [k] get_partial_node.isra.76
+    0.70%  [k] mlx5e_page_release
+    0.62%  [k] eth_type_trans
+    0.56%  [k] mlx5e_select_queue
     0.49%  [k] skb_gro_reset_offset
     0.42%  [k] skb_put
```

Packet rate 4574424 translates to ~219 nanosec (1/4574424*10^9).

The top contender is mlx5e_handle_rx_cqe(24.75%), which initially didn't surprise me, given I know that this function (via inlining) will be the first to touch the packet (via is_first_ethertype_ip()), thus causing a cache-line miss. **BUT something is wrong**. Looking at perf-annotate, the cache-line miss is NOT occurring, instead 67.24% CPU time spend on a refcnt increment (due to page_ref_inc(di->page) used for page-recycle cache). Something is wrong as 24.75% of 219 is 54ns, which is too high even for an atomic refcnt inc. (Note: the cache-miss is actually avoided due to the prefetch have time to work, due to this stall on the lock. Thus, removing the stall will bring-back the cache-line stall).

Inside __build_skb(16.95%) there is 83.47% CPU spend on "rep stos", which is clearing/memset-zero the SKB itself. Again something is wrong as ((1/4574424*10^9)*(16.95/100)) = 37ns is too high for clearing the SKB (time_bench_memset show this optimally takes 10 ns).

Inside mlx5e_xmit(10.72%) there is 17.96% spend on a sfence asm instruction. The cost (1/4574424*10^9)*(10.72/100) = 23.43 ns of calling mlx5e_xmit() might not be too off-target.

My guess is that this is caused the the tailptr/doorbell stall. And doing bulk/xmit_more we can likely reduce mlx5e_handle_rx_cqe(-12ns as cache-miss returns) and __build_skb(-27ns). Thus, the performance target should lay around 5.6Mpps ((1/(218-12-27)*10^9) = 5586592).

Also notice that __cmpxchg_double_slab() show that we are hitting the SLUB slow(er)-path.

Zooming into perf with Generic-netstack-XDP

Testing Generic-netstack-XDP_DROP again and looking closer at the perf reports. This will be intersting because we can deduct the cost of the different parts of the network stack, assuming there is no-fake stalls due to tailptr/doorbell (like the XDP_TX case)

```
[jbrouer@skylake prototype-kernel]$
 sudo ./xdp_bench01_mem_access_cost --action XDP_DROP --dev mlx5p2 --read
XDP_action   pps        pps-human-readable mem
XDP_DROP     8148835    8,148,835          read
XDP_DROP     8148972    8,148,972          read
XDP_DROP     8148962    8,148,962          read
XDP_DROP     8146856    8,146,856          read
XDP_DROP     8150026    8,150,026          read
XDP_DROP     8149734    8,149,734          read
XDP_DROP     8149646    8,149,646          read
```

For some unknown reason the Generic-XDP_DROP number are a bit lower, than above numbers. Using 8148972 pps (8,148,972) as our new baseline, show (averaged) cost per packet 122.47 nanosec (1/8165032*10^9)

The difference to NIC-level-XDP is: (1/12006685*10^9)- (1/8148972*10^9) = -39.42 ns

Simply perf recorded 30 sec, and find the CPU this was running on by added the –sort cpu to the output. The CPU output/column showed that NAPI was running on CPU 7

```
sudo ~/perf record -aR -g sleep 30
sudo ~/perf report --no-children  --sort cpu,comm,dso,symbol
```

Now we will drill down on CPU 7 and see what it is doing. We start with removing the “children” column, to start viewing the overhead on a per function basis.

I’m using this long perf report command to reduce the columns and print to stdout and removing the call graph (I’ll manually inspect the call-graph with the standard terminal-user-interface (TUI))

```
sudo ~/perf report --no-children  --sort symbol \
   --kallsyms=/proc/kallsyms -C7 --stdio -g none
```

Reduced output:

```
# Samples: 119K of event 'cycles'
# Event count (approx.): 119499252009
#
# Overhead  Symbol
# ........  ........................................
#
   34.33%  [k] mlx5e_handle_rx_cqe
   10.36%  [k] __build_skb
    5.49%  [k] build_skb
    5.10%  [k] page_frag_free
    4.06%  [k] bpf_prog_662b9cae761bf6ab
    4.02%  [k] kmem_cache_alloc
    3.85%  [k] netif_receive_skb_internal
    3.72%  [k] kmem_cache_free
    3.69%  [k] mlx5e_alloc_rx_wqe
    2.91%  [k] mlx5e_get_cqe
    1.83%  [k] napi_gro_receive
    1.80%  [k] __rcu_read_unlock
    1.65%  [k] skb_release_data
    1.49%  [k] dev_gro_receive
```

```
    1.43%  [k] skb_release_head_state
    1.26%  [k] mlx5e_post_rx_wqes
    1.22%  [k] mlx5e_page_release
    1.21%  [k] kfree_skb
    1.19%  [k] eth_type_trans
    1.00%  [k] __rcu_read_lock
    0.84%  [k] skb_release_all
    0.83%  [k] skb_free_head
    0.81%  [k] kfree_skbmem
    0.80%  [k] percpu_array_map_lookup_elem
    0.79%  [k] mlx5e_poll_rx_cq
    0.79%  [k] skb_put
    0.77%  [k] skb_gro_reset_offset
    0.63%  [k] swiotlb_sync_single
    0.61%  [k] swiotlb_sync_single_for_device
    0.42%  [k] swiotlb_sync_single_for_cpu
    0.28%  [k] net_rx_action
    0.21%  [k] bpf_map_lookup_elem
    0.20%  [k] mlx5e_napi_poll
    0.11%  [k] __do_softirq
    0.06%  [k] mlx5e_poll_tx_cq
    0.02%  [k] __raise_softirq_irqoff
```

Some memory observations are that we are hitting the fast path of the SLUB allocator (indicated by no func names from the slower path). The mlx5 driver-page recycler also have 100% hit rate, verified by looking at ethtool -S stats, and mlx5 stats "cache_reuse", using my ethtool_stats.pl tool:

```
Show adapter(s) (mlx5p2) statistics (ONLY that changed!)
Ethtool(mlx5p2) stat:     8179636 (      8,179,636) <= rx3_cache_reuse /sec
Ethtool(mlx5p2) stat:     8179632 (      8,179,632) <= rx3_packets /sec
Ethtool(mlx5p2) stat:    40657800 (     40,657,800) <= rx_64_bytes_phy /sec
Ethtool(mlx5p2) stat:   490777805 (    490,777,805) <= rx_bytes /sec
Ethtool(mlx5p2) stat:  2602103605 (  2,602,103,605) <= rx_bytes_phy /sec
Ethtool(mlx5p2) stat:     8179636 (      8,179,636) <= rx_cache_reuse /sec
Ethtool(mlx5p2) stat:     8179630 (      8,179,630) <= rx_csum_complete /sec
Ethtool(mlx5p2) stat:    18736623 (     18,736,623) <= rx_discards_phy /sec
Ethtool(mlx5p2) stat:    13741170 (     13,741,170) <= rx_out_of_buffer /sec
Ethtool(mlx5p2) stat:     8179630 (      8,179,630) <= rx_packets /sec
Ethtool(mlx5p2) stat:    40657861 (     40,657,861) <= rx_packets_phy /sec
Ethtool(mlx5p2) stat:  2602122863 (  2,602,122,863) <= rx_prio0_bytes /sec
Ethtool(mlx5p2) stat:    21921459 (     21,921,459) <= rx_prio0_packets /sec
[...]
```

Knowing the cost per packet 122.47 nanosec (1/8165032*10^9), we can extrapolate the ns used by each function call. Let use oneline for calculating that for us:

```
sudo ~/perf report --no-children  --sort symbol \
  --kallsyms=/proc/kallsyms -C7 --stdio -g none | \
awk -F% 'BEGIN {base=(1/8165032*10^9)} \
  /%/ {ns=base*($1/100); \
       printf("%6.2f\% => %5.1f ns func:%s\n",$1,ns,$2);}'
```

Output:

```
34.33% =>  42.0 ns func:  [k] mlx5e_handle_rx_cqe
10.36% =>  12.7 ns func:  [k] __build_skb
 5.49% =>   6.7 ns func:  [k] build_skb
 5.10% =>   6.2 ns func:  [k] page_frag_free
```

```
 4.06% =>   5.0 ns func:  [k] bpf_prog_662b9cae761bf6ab
 4.02% =>   4.9 ns func:  [k] kmem_cache_alloc
 3.85% =>   4.7 ns func:  [k] netif_receive_skb_internal
 3.72% =>   4.6 ns func:  [k] kmem_cache_free
 3.69% =>   4.5 ns func:  [k] mlx5e_alloc_rx_wqe
 2.91% =>   3.6 ns func:  [k] mlx5e_get_cqe
 1.83% =>   2.2 ns func:  [k] napi_gro_receive
 1.80% =>   2.2 ns func:  [k] __rcu_read_unlock
 1.65% =>   2.0 ns func:  [k] skb_release_data
 1.49% =>   1.8 ns func:  [k] dev_gro_receive
 1.43% =>   1.8 ns func:  [k] skb_release_head_state
 1.26% =>   1.5 ns func:  [k] mlx5e_post_rx_wqes
 1.22% =>   1.5 ns func:  [k] mlx5e_page_release
 1.21% =>   1.5 ns func:  [k] kfree_skb
 1.19% =>   1.5 ns func:  [k] eth_type_trans
 1.00% =>   1.2 ns func:  [k] __rcu_read_lock
 0.84% =>   1.0 ns func:  [k] skb_release_all
 0.83% =>   1.0 ns func:  [k] skb_free_head
 0.81% =>   1.0 ns func:  [k] kfree_skbmem
 0.80% =>   1.0 ns func:  [k] percpu_array_map_lookup_elem
 0.79% =>   1.0 ns func:  [k] mlx5e_poll_rx_cq
 0.79% =>   1.0 ns func:  [k] skb_put
 0.77% =>   0.9 ns func:  [k] skb_gro_reset_offset
 0.63% =>   0.8 ns func:  [k] swiotlb_sync_single
 0.61% =>   0.7 ns func:  [k] swiotlb_sync_single_for_device
 0.42% =>   0.5 ns func:  [k] swiotlb_sync_single_for_cpu
 0.28% =>   0.3 ns func:  [k] net_rx_action
 0.21% =>   0.3 ns func:  [k] bpf_map_lookup_elem
 0.20% =>   0.2 ns func:  [k] mlx5e_napi_poll
 0.11% =>   0.1 ns func:  [k] __do_softirq
```

top contender mlx5e_handle_rx_cqe

The top contender mlx5e_handle_rx_cqe() in the driver code

```
34.33% =>  42.0 ns func:  [k] mlx5e_handle_rx_cqe
```

When looking at the code/perf-annotate do notice that several function calls have been inlined by the compiler. The thing that light-up (56.23% => 23.6 ns) in perf-annotate is touching/reading the data-packet for the first time, which is reading the ethertype via is_first_ethertype_ip(), called via:

- which is called from mlx5e_handle_csum()
- which is called by mlx5e_build_rx_skb()
- which is called by mlx5e_complete_rx_cqe()
- which is called by mlx5e_handle_rx_cqe() all inlined.

Notice this is_first_ethertype_ip() call is the reason why eth_type_trans() is not so hot in this driver.

Analyzing __build_skb and memset

The compiler choose not to inline __build_skb(), and what is primarily going on here is memset clearing the SKB data, which gets optimized into an "rep stos" asm-operation, which is actually not optimal for this size of objects.

Looking at perf-annotate shows that 75.65% of the time of __build_skb() is spend on "rep stos %rax,%es:(%rdi)". Thus, extrapolating 12.7 ns (12.7*(75.65/100)) cost of 9.6 ns.

This is very CPU specific how fast or slow this is, but I've benchmarked different alternative approaches with time_bench_memset.c.

Memset benchmarks on this Skylake CPU show that hand-optimizing ASM-coded memset, can reach 8 bytes per cycles, but only saves approx 2.5 ns or 10 cycles. A more interesting approach would be if we could memset clear a larger area. E.g. when bulk-allocating SKBs and detecting they belong to the same page and is contiguous in memory. Benchmarks show that clearing larger areas is more efficient.

Table with memset "rep-stos" size vs bytes-per-cycle efficiency

```
$ perl -ne 'while(/memset_(\d+) .* elem: (\d+) cycles/g)\
   {my $bpc=$1/$2; \
    printf("memset %5d bytes cost %4d cycles thus %4.1f bytes-per-cycle\n", \
           $1, $2, $bpc);}' memset_test_dmesg

memset    32 bytes cost    4 cycles thus  8.0 bytes-per-cycle
memset    64 bytes cost   29 cycles thus  2.2 bytes-per-cycle
memset   128 bytes cost   29 cycles thus  4.4 bytes-per-cycle
memset   192 bytes cost   35 cycles thus  5.5 bytes-per-cycle
memset   199 bytes cost   35 cycles thus  5.7 bytes-per-cycle
memset   201 bytes cost   39 cycles thus  5.2 bytes-per-cycle
memset   204 bytes cost   40 cycles thus  5.1 bytes-per-cycle
memset   200 bytes cost   39 cycles thus  5.1 bytes-per-cycle
memset   208 bytes cost   39 cycles thus  5.3 bytes-per-cycle
memset   256 bytes cost   36 cycles thus  7.1 bytes-per-cycle
memset   512 bytes cost   40 cycles thus 12.8 bytes-per-cycle
memset   768 bytes cost   47 cycles thus 16.3 bytes-per-cycle
memset  1024 bytes cost   52 cycles thus 19.7 bytes-per-cycle
memset  2048 bytes cost   84 cycles thus 24.4 bytes-per-cycle
memset  4096 bytes cost  148 cycles thus 27.7 bytes-per-cycle
memset  8192 bytes cost  276 cycles thus 29.7 bytes-per-cycle
```

I've already implemented the SLUB bulk-alloc API, and it could be extended with detecting if objects are physically contiguous for allowing clearing multiple object at the same time. (Notice the SLUB alloc-side fast-path already delivers object from the same page).

Blaming the children

The nanosec number are getting so small, that we might miss the effect of deep call chains. Thus, lets look at perf report with the "children" enabled:

```
 Samples: 119K of event 'cycles', Event count (approx.): 119499252009
  Children      Self  Symbol
+  100.00%     0.00%  [k] kthread
+  100.00%     0.00%  [k] ret_from_fork
+   99.99%     0.01%  [k] smpboot_thread_fn
+   99.98%     0.01%  [k] run_ksoftirqd
+   99.94%     0.11%  [k] __do_softirq
+   99.78%     0.28%  [k] net_rx_action
+   99.41%     0.20%  [k] mlx5e_napi_poll
+   92.44%     0.79%  [k] mlx5e_poll_rx_cq
+   86.37%    34.33%  [k] mlx5e_handle_rx_cqe
+   29.40%     1.83%  [k] napi_gro_receive
+   24.50%     3.85%  [k] netif_receive_skb_internal
+   19.41%     5.49%  [k] build_skb
```

```
+   14.98%     1.21%  [k] kfree_skb
+   14.15%    10.36%  [k] __build_skb
+    9.43%     0.84%  [k] skb_release_all
+    6.97%     1.65%  [k] skb_release_data
+    5.38%     1.26%  [k] mlx5e_post_rx_wqes
+    5.10%     5.10%  [k] page_frag_free
+    4.86%     4.06%  [k] bpf_prog_662b9cae761bf6ab
+    4.30%     3.69%  [k] mlx5e_alloc_rx_wqe
+    4.30%     0.81%  [k] kfree_skbmem
+    4.02%     4.02%  [k] kmem_cache_alloc
+    3.72%     3.72%  [k] kmem_cache_free
+    2.91%     2.91%  [k] mlx5e_get_cqe
```

Lets calculate the ns cost:

```
 $ sudo ~/perf report --children  --sort symbol \
   --kallsyms=/proc/kallsyms -C7 --stdio -g none | \
   awk -F% 'BEGIN {base=(1/8165032*10^9); \
            print "Children => nanosec     Self    Symbol/fucntion\n";} \
     /%/ {ns=base*($1/100); \
         printf("%6.2f%s => %5.1f ns %s%s func:%s\n",$1,"%",ns,$2,"%",$3);}'

Children => nanosec     Self    Symbol/fucntion
100.00% => 122.5 ns      0.00% func:  [k] kthread
100.00% => 122.5 ns      0.00% func:  [k] ret_from_fork
 99.99% => 122.5 ns      0.01% func:  [k] smpboot_thread_fn
 99.98% => 122.4 ns      0.01% func:  [k] run_ksoftirqd
 99.94% => 122.4 ns      0.11% func:  [k] __do_softirq
 99.78% => 122.2 ns      0.28% func:  [k] net_rx_action
 99.41% => 121.8 ns      0.20% func:  [k] mlx5e_napi_poll
 92.44% => 113.2 ns      0.79% func:  [k] mlx5e_poll_rx_cq
 86.37% => 105.8 ns     34.33% func:  [k] mlx5e_handle_rx_cqe
 29.40% =>  36.0 ns      1.83% func:  [k] napi_gro_receive
 24.50% =>  30.0 ns      3.85% func:  [k] netif_receive_skb_internal
 19.41% =>  23.8 ns      5.49% func:  [k] build_skb
 14.98% =>  18.3 ns      1.21% func:  [k] kfree_skb
 14.15% =>  17.3 ns     10.36% func:  [k] __build_skb
  9.43% =>  11.5 ns      0.84% func:  [k] skb_release_all
  6.97% =>   8.5 ns      1.65% func:  [k] skb_release_data
  5.38% =>   6.6 ns      1.26% func:  [k] mlx5e_post_rx_wqes
  5.10% =>   6.2 ns      5.10% func:  [k] page_frag_free
  4.86% =>   6.0 ns      4.06% func:  [k] bpf_prog_662b9cae761bf6ab
  4.30% =>   5.3 ns      3.69% func:  [k] mlx5e_alloc_rx_wqe
  4.30% =>   5.3 ns      0.81% func:  [k] kfree_skbmem
  4.02% =>   4.9 ns      4.02% func:  [k] kmem_cache_alloc
  3.72% =>   4.6 ns      3.72% func:  [k] kmem_cache_free
  2.91% =>   3.6 ns      2.91% func:  [k] mlx5e_get_cqe
  1.80% =>   2.2 ns      1.80% func:  [k] __rcu_read_unlock
  1.49% =>   1.8 ns      1.49% func:  [k] dev_gro_receive
  1.43% =>   1.8 ns      1.43% func:  [k] skb_release_head_state
  1.22% =>   1.5 ns      1.22% func:  [k] mlx5e_page_release
  1.19% =>   1.5 ns      1.19% func:  [k] eth_type_trans
  1.00% =>   1.2 ns      1.00% func:  [k] __rcu_read_lock
  0.84% =>   1.0 ns      0.83% func:  [k] skb_free_head
  0.80% =>   1.0 ns      0.80% func:  [k] percpu_array_map_lookup_elem
  0.79% =>   1.0 ns      0.79% func:  [k] skb_put
  0.77% =>   0.9 ns      0.77% func:  [k] skb_gro_reset_offset
```

Interesting here is napi_gro_receive() which is the base-call into the network stack, everything "under" this call cost 29.40% of the time, translated to 36.0 ns. This 36 ns cost is interesting as we calculated the difference to NIC-level-XDP to be 39 ns:

The difference to NIC-level-XDP is: (1/12006685*10^9)- (1/8148972*10^9) = -39.42 ns

Freeing the SKB is summed up under kfree_skb() with 14.98% => 18.3 ns. In this case kfree_skb() should get attributed under napi_gro_receive(), due to the direct kfree_skb(skb) call in netif_receive_generic_xdp(). In other situations kfree_skb() happens during the DMA TX completion, but not here.

Creating, allocating and clearing the SKB is all "under" the build_skb() call, which attributes to a collective 19.41% or 23.8 ns. The build_skb() call happens, in-driver, before calling napi_gro_receive.

Thus, one might be lead to conclude that the overhead of the network stack is (23.8 ns +36 ns) 59.8 ns, but something is not adding up as this is higher the calculated approx 40ns difference to NIC-level-XDP.

CHAPTER 7

Indices and tables

- genindex
- search