The Challenges of XDP Hardware Offload

Quentin Monnet
<quentin.monnet@netronome.com>
@qeole

NETRONOME
eBPF and XDP
eBPF, extended Berkeley Packet Filter

Generic, efficient, secure in-kernel (Linux) virtual machine
Programs are injected and attached in the kernel, event-based
eBPF architecture

- Evolution from former BPF version (cBPF, used by tcpdump)
- Assembly-like instructions, 4096 maximum in a program
- 11 registers (64-bit), 512 bytes stack
- Read and write access to context (for networking: packets)
- LLVM backend to compile from C to eBPF (or from Lua, Go, P4, Rust, ...)
- In-kernel verifier to ensure safety, security
- JIT (Just-in-time) compiler available for main architectures
- Programs managed with bpf() system call, loaded with e.g. tc, ip
**eBPF features**

- **Maps**: key-value entries (hash, array, ...), shared between eBPF programs or with user space.

- **Tail calls**: “long jump” from one program into another, context is preserved.

- **Helpers**: white-list of kernel functions to call from eBPF programs: get current time, print debug information, lookup or update maps, shrink or grow packets, ...
eBPF workflow

- **C source code**: bpf_prog.c
- **ELF-compiled eBPF**: bpf_prog.o
- **LLVM/clang**
- **Verifier**
- **User program** (tc / ip / bcc tools...)
- **User program**
- **Maps**
  - Array
  - Hashmap
  - LPM
  - ...
- **BPF program attached and run**
- **JIT**
- **Userspace**
- **Kernel**
Introduced in Linux 4.8

eBPF hook at the driver level (ingress)
Intercept packet before it reaches the stack, before allocating sk_buff

Rationale: implement a faster data path which is part of the kernel, maintained by the kernel community

Rather for simple use cases. Complex processing: forward to stack

Not a “kernel bypass”, works in cooperation with the networking stack
XDP architecture

- Userspace
- Kernel
- Network stack
  - tc ingress
  - tc egress
- Net device
- Sockets
- Packets
XDP architecture

- **Userspace**
- **Kernel**
- **Network stack**
  - `tc ingress`
  - `tc egress`
- **Packets**
- **Sockets**
- **Redirect to any device**
- **Drop (or Abort)**
- **Edit and bounce**
- **Forward to stack**

**DPDK**
Use cases for eBPF and XDP

- Load balancing
- Protection, mitigation against DDoS
- Distributed firewall
- And a lot more
  - Packet capture (Suricata)
  - Network fabric (OVN), Container ACLs (Cilium)
  - Virtual switching: Open vSwitch back-end
  - Stateful processing (BEBA research project)
  - ILA (Identifier-Locator Addressing) routing
  - QoS
  - ...
eBPF Hardware Offload
Why offloading to hardware?

- eBPF is nearly “self-contained”, XDP is low-level: ideal for offload
- Get performances, and get programmability — without putting the charge on CPUs
- Work with the kernel: push hardware offload support upstream
  Still requires NIC and firmware, but make driver and eBPF core available to the community
1. Get the correct architecture
Mapping eBPF to the NFP-based NIC

- 11 Registers (64 bit, 32 bit subregisters)
- Stack (512 bytes)
- Maps (varying size)
- Driver
- General Purpose Registers
- LMEM (1 kB)
- Thread (4 per core)
- Core (60 used for eBPF)
- CTM – Cluster Target Memory (256 kB)
- CLS – Cluster Local Scratch (64 kB)
- IMEM (4 MB)
- DRAM (2 GB)

Q. Monnet | XDP Hardware Offload
How to get a program we can run?

- The driver has its own JIT, called by the kernel, and compiles to native instructions for the NIC.
- NIC has 32-bit registers: eBPF 32-bit support in the kernel
- Various optimisations in the JIT to reduce the number of instructions or speed up some tasks
The Challenges of eBPF Hardware Offload

1. Get a compatible architecture
   - NIC architecture
   - Add 32-bit support for eBPF
   - Use own JIT-compiler

2. Add offload support to the kernel
Support for eBPF offload in kernel and driver

- C code
- ELF-compiled eBPF

bpf() syscall

- program instructions
- program length
- type (cls, XDP, kprobe...)
- ...

Verifier
- Verification
- Modification

eBPF program
- JIT

User program (tc / ip / bcc tools...)
- file descriptor

Userspace
- TC cls_bpf

Kernel
Support for eBPF offload in kernel and driver

- **bpf() syscall**
  - Find device
  - Verifier
    - Verification
    - Modification
  - JIT

- **User program**
  - (tc / ip / bcc tools...)
  - file descriptor
  - program instructions
  - program length
  - type (cls, XDP, kprobe...)
  - ifindex

- **User program**
  - Offload object
  - XDP
  - Offload object
  - XDP

- **Driver operations**
  - XDP attach
  - Prepare verifier
  - Check instruction
  - Translate
  - Destroy

- **TC cls_bpf**

- **Driver**
  - RX
  - XDP
  - TX

- **ndo_setup_tc()**

C code → ELF-compiled eBPF

Q. Monnet | XDP Hardware Offload
Support for eBPF offload in kernel and driver

- **bpf() syscall**
  - Find device
    - Verifier
      - Verification
      - Modification
    - JIT
  - ELF-compiled eBPF
    - program instructions
    - program length
    - type (cls, XDP, kprobe...)
    -...
    - ifindex
    - file descriptor
  - User program (tc / ip / bcc tools...)
  - User program (tc / ip / bcc tools...)
  - Driver operations
    - XDP attach
    - Prepare verifier
    - Check instruction
    - Translate
    - Destroy
    - ndo_bpf()
    - ndo_setup_tc()

- **TC cls_bpf**
  - Offload object

- **Userspace**
  - User program
  - Driver
  - RX
  - XDP
  - TX
Some notes on eBPF offload support in the kernel

- The verifier uses a callback to check each instruction from the driver perspective

- The driver has its own errors that we must expose to users:
  - Verification time: reuse the log buffer from kernel verifier → STD_ERR
  - Program attachment time: use Netlink extended ack → STD_ERR
The Challenges of eBPF Hardware Offload

1. Get a compatible architecture
   - NIC architecture
   - Add 32-bit support for eBPF
   - Use own JIT-compiler

2. Add offload support to the kernel
   - Update verifier
   - Make the core able to pass eBPF maps and programs
   - Keep it human-friendly

3. Update the tools
Upgrade tools for handling offloaded programs (tc, ip)
- Update command syntax
- Pass the ifindex to the kernel
- Also ask kernel to create maps on the NIC

Create or update other tools to help working with eBPF
- bpftool
  - List, load, pin, dump instructions (JIT-ed or not) for programs
  - List, pin, dump, lookup, update, delete for maps
  - List, attach, detach programs to cgroups
- llvm-mc: Compile from “eBPF assembly” to object file
The Challenges of eBPF Hardware Offload

1. Get a compatible architecture
   - NIC architecture
   - Add 32-bit support for eBPF
   - Use own JIT-compiler

2. Add offload support to the kernel
   - Update the verifier
   - Make the core able to pass eBPF maps and programs
   - Keep it human-friendly

3. Update the tools
   - tc, ip, llvm-mc, bpftool

4. Gain better performances, everywhere you can!
What we have

- tc_cls and XDP hardware offload (specific JIT)
- 32-bit sub-registers support
- Various JIT optimisations
- Nearly all instructions supported; Stack; Some helpers
- Direct packet access, packet modification (header or payload)
- XDP actions: Bounce, Pass to stack, Drop; Packet encapsulation
- Maps: hashes and arrays (RO from program, R/W from user space)
- Error messages through integration with kernel verifier, extack

Tooling
  - tc, ip updated
  - bpftool
  - llvm-mc
Performances

- Simple XDP load balancer (~ 800 eBPF insns, 4 map lookups)
  - Based on kernel test tools/testing/selftests/bpf/test_l4lb.c, combined with example samples/bpf/xdp_tx_iptunnel_kern.c

- Per CPU array changed to standard array to run offloaded
  - (No nice equivalent for per CPU at the moment on the NIC)

Sample Load Balancer
NFP can viably offload applications in XDP-and lots of performance headroom

![Performance Chart]

- XDPOFFLOAD (1 Core) Optimized Maps
- XDPOFFLOAD (1 Core)
- XDPDRV
- XDPDRV/Core
What we work on

- Redirect action
- Atomic add operation
- Map caching: map access from ~1000 to ~300 cycles
- Packet caching: packet accesses from ~50 to ~3 cycles
- 32-bit ALU from LLVM where possible: ALUs from ~4 to 1 machine code instruction
- Remove firmware locks for maps: double memory bandwidth
- Tail calls; Multi-stage processing, split between NIC and host

- Dump NFP instructions with bpftool: need patching binutils-dev
- More JIT optimisations
- ...
The Challenges of eBPF Hardware Offload

- eBPF and XDP introduce fast and efficient networking inside Linux kernel

- Host CPU is a resource and must be used efficiently. Getting faster networking without increasing CPU usage requires an efficient and transparent general offload infrastructure in cooperation with the kernel

- eBPF, XDP offload provides programmability and performances, but also a dynamically reloadable sandbox

- Kernel, driver: everything is upstream!
Thank you!

Questions?

Additional resources:

Open-NFP.org platform, with resources about eBPF offload
https://open-nfp.org/dataplanes-ebpf/

Resources on BPF — *Dive into BPF: a list of reading material*
https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/

Upstream driver, eBPF bits
Linux kernel tree, under drivers/net/ethernet/netronome/nfp/bpf

Netronome website
https://www.netronome.com/

*We’re hiring!*