DPDK+eBPF

KONSTANTIN ANANYEV
INTEL

BPF overview

- BPF (Berkeley Packet Filter) is a VM in the kernel (linux/freebsd/etc.) allowing to execute bytecode at various hook points in a safe manner.
- It is used in a number of Linux kernel subsystems:
  - networking
    - Socket filtering for most protocols
    - tc classifier (cls bpf)
    - netfilter xtables (xt bpf)
    - XDP
  - tracing
    - BPF as kprobes-based extensions
  - etc…

BPF Architecture

- Classic BPF (cBPF)
  - 32 bit, available register: A, X, M[0-15], (pc)
  - Forward jumps only, max 4096 instructions
  - JIT for all major archs
- Extended BPF (eBPF)
  - eleven 64 bit registers with 32 bit sub-registers, a program counter and 512B stack
  - 64 bit instruction format (u8:code, u8:dst reg, u8:src reg, s16:off, s32:imm)
  - New insns: dw ld/st, mov, alu64 + signed shift, endian, calls, xadd
  - Forward (& backward) jumps, max 4096 instructions
  - Generic helper function concept, several kernel-provided helpers
  - Maps with arbitrary sharing (user space apps, between eBPF progs)
  - clang eBPF backend (v3.7 or above)
  - HW offload

BPF within DPDK

Might also be used in a lot of places to help with:

- packet filtering/tracing (aka tcpdump)
- packet classification
- statistics collection
- HW/PMD live-system debugging/prototyping - trace HW descriptors, internal PMD SW state
- etc…

All of that in a dynamic, user-defined and extensible manner.

Current status

librte_bpf integrated into DPDK 18.05

- Supported features:
  - base eBPF ISA (except tail-pointer)
  - JIT (x86_64 only)
  - eBPF code verifier
  - user-defined helper functions (64-bit only)
  - RX/TX filter (ability to load/execute eBPF program as ethdev RX/TX call-back, no need to stop/start ethdev port/queue)
  - rte_mbuf access (64-bit only)

- Currently not supported features:
  - cBPF
  - MAPs
  - tail-pointer calls

librte_bpf API

- Generic API:

```
struct rte_bpf * rte_bpf_load(const struct rte_bpf_prm *prm);

struct rte_bpf * rte_bpf_elf_load(const struct rte_bpf_prm *prm, const char *fname, const char *sname);

void rte_bpf_destroy(struct rte_bpf *bpf);

uint64_t rte_bpf_exec(const struct rte_bpf *bpf, void *ctx);

uint32_t rte_bpf_exec_burst(const struct rte_bpf *bpf, void *ctx[], uint64_t rc[], uint32_t num);

int rte_bpf_get_jit(const struct rte_bpf *bpf, struct rte_bpf_jit *jit);
```

- RX/TX filter API:

```
int rte_bpf_eth_rx_elf_load(uint16_t port, uint16_t queue, const struct rte_bpf_prm *prm, const char *fname,
                            const char *sname, uint32_t flags);

int rte_bpf_eth_rx_elf_load(uint16_t port, uint16_t queue, const struct rte_bpf_prm *prm, const char *fname,
                            const char *sname, uint32_t flags);

void rte_bpf_eth_rx_unload(uint16_t port, uint16_t queue);

void rte_bpf_eth_rx_unload(uint16_t port, uint16_t queue);
```

How to try it

1. run testpmd as usual and start your favorite forwarding case.
2. build bpf program you'd like to load:

   $ cd test/bpf

   $ clang -O2 -target bpf -c t1.c
3. load/unload bpf program:

   testpmd> bpf-load rx|tx <portid> <queueid> <load-flags> <filename>
   testpmd> bpf-unload rx|tx <portid> <queueid>

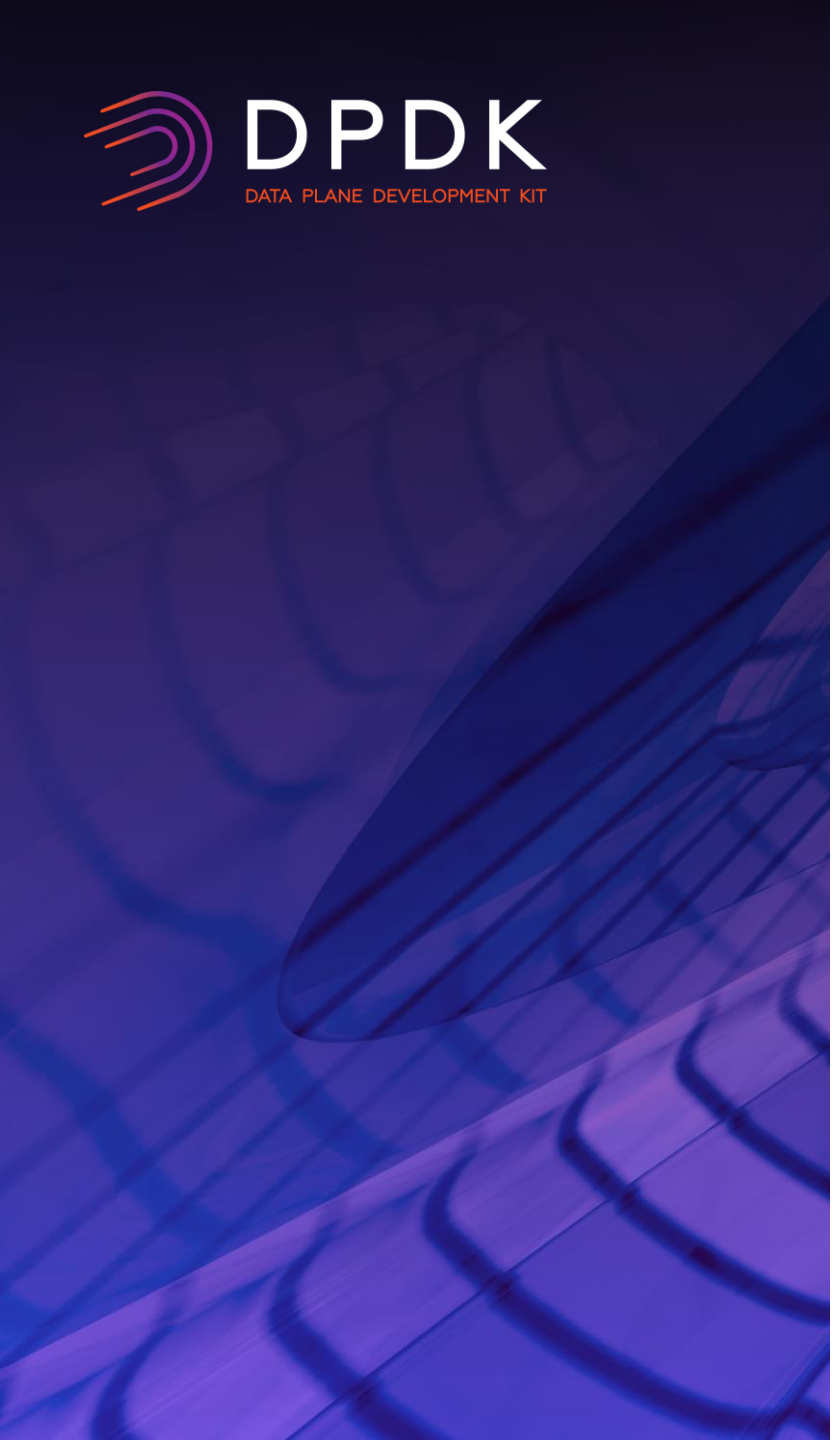
   As an example:
   - #to load (and JIT compile) t1.o at RX queue 0, port 1:
     testpmd> bpf-load rx 1 0 J ./dpdk.org/test/bpf/t1.o
   - #to unload t1.o and load and JIT t3.o (note that it expects mbuf as an input):
     testpmd> bpf-load rx 1 0 JM ./dpdk.org/test/bpf/t3.o

- $ cat t1.c

```
/*
 * eBPF program sample.
 * Accepts pointer to first segment packet data as an input parameter.
 * analog of tcpdump -s 1 -d 'dst 1.2.3.4 && udp && dst port 5000'
 */
#include <stdint.h>
#include <net/ethernet.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
uint64_t
entry(void *pkt)
{
        struct ether_header *ether_header = (void *)pkt;
        if (ether_header->ether_type != __builtin_bswap16(0x0800))
                return 0;
        struct iphdr *iphdr = (void *)(ether_header + 1);
        if (iphdr->protocol != 17 || (iphdr->frag_off & 0x1fff) != 0 ||
                        iphdr->daddr != __builtin_bswap32(0x1020304))
                return 0;
    int hlen = iphdr->ihl * 4;
     struct udphdr *udphdr = (void *)iphdr + hlen;
      if (udphdr->dest !=  __builtin_bswap16(5000))
                return 0;
          return 1;
}
```

Possible future development

- Add cBPF support.
- Add JIT for other architectures.
- Improve a verifier.
- Bulk version of JIT code.
- Performance improvements.
- Add MAP support.
- HW offload
- …?



Q&A