Open-NFP Summer Webinar Series: Session 4: P4, EBPF And Linux TC Offload

Dinan Gunawardena & Jakub Kicinski - Netronome
August 24, 2016
Support and grow reusable research in accelerating dataplane network functions processing

Reduce/eliminate the cost and technology barriers to research in this space

- **Technologies:**
  - P4, eBPF, SDN, OpenFlow, Open vSwitch (OVS) offload

- **Tools:**
  - Discounted hardware, development tools, software, cloud access

- **Community:**
  - Website (www.open-nfp.org): learning & training materials, active Google group [https://groups.google.com/d/forum/open-nfp](https://groups.google.com/d/forum/open-nfp), open project descriptions, code repository

- **Learning/Education/Research support:**
  - Summer seminar series, P4DevCon conference, Tutorials (P4 Developer Day), research proposal support for proposals to the NSF, state agencies

Summer seminar series to further progress to our objective. Present applied reusable research.
P4DevCon Attendees/Open-NFP Projects*

Universities

Companies

*This does not imply that these organizations endorse Open-NFP or Netronome
Session Agenda

1. Introduction
   - Objectives

2. Overview: The high level model for offload
   - What we are offloading – P4 / eBPF
   - Overall programmer model for transparent offload

3. Linux Kernel Infrastructure
   - The Traffic Classifier (TC)
   - eXpress Data Path (XDP)
   - Current eBPF translation on X86/ARM64/PPC64

4. Hardware Intro to NFP (Network Flow Processor) architecture
   - SmartNICs-Multi Core, Many Core
   - NUMA, Memory Hierarchy

5. Accelerating P4/eBPF in NFP: Implementation
   - Kernel core infrastructure
   - Map handling in the kernel
   - Translating instructions
   - Basic Map Support
   - Optimizations

6. & 7. Demo; Summary
Session Objectives
Introduction: Objectives

Understanding how eBPF is relevant to P4

Understanding the support for offload and state of art in the Linux Kernel

The Code
- Understand the structure of a eBPF program
- Gain an insight into how this is translated and executed in hardware

Understanding how the NFP architecture on the Agilio CX enables high performing, fully programmable network offload
- The Many Core architecture and its advantages
Overview: High level model for offload

- What we are offloading – P4 / eBPF
- Overall programmer model for transparent offload
P4 and eBPF

What are P4 and eBPF?
Domain specific languages for specifying forwarding behaviour of the data plane of network components

P4 - Programming Protocol-Independent Packet Processors
• Header format description
• Parse Graphs (FSM)
• Tables (<keys,actions>)
• Actions manipulate packet header/metadata
• Control flow – an imperative program, describing sequence of data dependent match/actions

eBPF – Extended Berkley Packet Filters
• Low level (machine code like) language
• Code executed by a VM (restricted memory, no sleeps/locks, limited API to kernel) in the Kernel (TC)
• Code injected into netfilter hook points in kernel data plane
• Maps (<key, value> stores)
• Chained filter functions
• Match/action
• Static verification of safety, guaranteed to terminate
Translating P4->eBPF

John Fastabend P4 to eBPF compiler

Why translate P4 to eBPF?

<table>
<thead>
<tr>
<th>P4 Construct</th>
<th>C Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>header_type</td>
<td>struct type</td>
</tr>
<tr>
<td>header</td>
<td>struct instance with an additional valid bit</td>
</tr>
<tr>
<td>metadata</td>
<td>struct instance</td>
</tr>
<tr>
<td>parser state</td>
<td>code block</td>
</tr>
<tr>
<td>state transition</td>
<td>goto statement</td>
</tr>
<tr>
<td>extract</td>
<td>load/shift/mask data from packet buffer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P4 Construct</th>
<th>C Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>2 EBPF tables: second one used just for the default action</td>
</tr>
<tr>
<td>table key</td>
<td>struct type</td>
</tr>
<tr>
<td>table actions block</td>
<td>tagged union with all possible actions</td>
</tr>
<tr>
<td>action arguments</td>
<td>struct</td>
</tr>
<tr>
<td>table reads</td>
<td>EBPF table access</td>
</tr>
<tr>
<td>action body</td>
<td>code block</td>
</tr>
<tr>
<td>table apply</td>
<td>switch statement</td>
</tr>
<tr>
<td>counters</td>
<td>additional EBPF table</td>
</tr>
</tbody>
</table>
Model for Transparent Offload

Programmer/user is “unaware” that eBPF code is “offloaded”

Requirements

• Partial pipeline offload
• Fallback to software for any eBPF code block
• Transparent user mode/kernel mode access to tables

Packet In → eBPF Program 1 (HW offload) → eBPF Program 2 (HW offload) → PCI-E crossing → eBPF Program 3
Linux Kernel Infrastructure

- The Traffic Classifier (TC)
- eXpress Data Path (XDP)
Linux Traffic Classifier (TC)

<table>
<thead>
<tr>
<th>Component</th>
<th>Linux Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shaping</td>
<td>The class offers shaping capabilities</td>
</tr>
<tr>
<td>Scheduling</td>
<td>A qdisc acts as a scheduler e.g. FIFO</td>
</tr>
<tr>
<td>Classifying</td>
<td>The filter performs classification through a classifier object.</td>
</tr>
<tr>
<td>Policing</td>
<td>A policer performs policing within a filter</td>
</tr>
<tr>
<td>Dropping</td>
<td>The “drop” action occurs with a filter+policer</td>
</tr>
<tr>
<td>Marking</td>
<td>The dsmark qdisc is used for marking</td>
</tr>
</tbody>
</table>

**A Simple Program**

TC diagram and example program © Jamal Hadi Salim
eXpress Data Path (XDP)

What
• High performance, programmable network data path

Utility
• Useful for packet processing
• forwarding
• load balancing
• DOS mitigation
• firewalls, etc.
Hardware Intro to NFP (Network Flow Processor) architecture

- NUMA, Memory Hierarchy
- Current eBPF translation on X86/ARM64/PPC64
NUMA, Memory Hierarchy

Architectural Philosophies:

- Bring the data close to where it needs to be processed
- Facilitate highly concurrent access to memories
- Mitigate branch costs and hide I/O & memory access latencies
Current eBPF translation on X86/ARM64/PPC64

1) Write eBPF program as a simple C Program
2) Compiled to eBPF byte code
3) Loaded into the Linux TC
4) Run through verifier
5) Cross compiled to X86/ARM64/PPC64

… Or now NFP Byte Code!
Dataflow

User Space

TCP Stack

Traffic Classifier (TC)

Driver (XDP)

Firmware & Hardware

Packet + Sk_Buf

Packet + Descriptor, Meta data

User

Kernel

PCI-E

NFP

Network
Supported Actions

- Drop
- Mark
- Redirect

- User Space
- Traffic Classifier (TC)
- Driver (XDP)
- Firmware & Hardware

Packet + Sk_Buf
Packet + Descriptor, Meta data

User
Kernel
PCI-E
NFP
Network
Accelerating P4/eBPF in NFP: Implementation

- Kernel core infrastructure
- Map handling in the kernel
- eBPF to NFP
- Map Support
- Optimizations
Kernel core infrastructure

skip_sw | skip_hw

- eBPF Program + Flags
  - NFP Offload Control
    - Offload obj
      - Stats
    - Firmware & Hardware
  - User Space
    - TCP Stack
      - Traffic Classifier (TC)
        - Driver (XDP)
          - Stats

User
Kernel
PCI-E
NFP
Network

©2016 Open-NFP
eBPF to NFP

- Non-linear mapping
- 32 bit translation
Map Handling in Kernel & Map Write Reflection

- Write interception
- Ownership
- Associating with a device

Write Reflection

- Read-Only single-user maps
- Read-Only multi-user maps
- Write-Enabled single-user maps
- Write-Enabled multi-user maps
Optimizations

• Dealing with different memory types

• Peephole Optimizer
  • Dropping unnecessary jumps
  • Optimizing instructions with immediates
  • Fusing instructions

• Full A/B register allocations

• Liveness analysis with real state size tracking
Demo
Summary

• Learnt the relationship between P4 and eBPF
• Discovered the infrastructure in the Linux Kernel for eBPF offload
• Learnt about how the Netronome Smart NIC architecture is optimised for network flow processing
• Explored how we implemented the eBPF offload in hardware
QUESTIONS?

Dinan Gunawardena
dinan.gunawardena@netronome.com

Jakub Kicinski
Jakub.Kicinski@netronome.com
THANK YOU