Towards a Stateful Forwarding Abstraction to Implement Scalable Network Functions in Software and Hardware

Luca Petrucci‡, Nicola Bonelli*, Marco Bonola‡, Gregorio Procissi*, Carmelo Cascone+, Davide Sanvito+, Salvatore Pontarelli‡, Giuseppe Bianchi‡, Roberto Bifulco†

* CNIT/University of Pisa ‡ CNIT/University of Rome Tor Vergata + CNIT/Politecnico di Milano † NEC Laboratories Europe

Italian Networking Workshop (INW)
Falcade, 11-13 January 2017

Work in progress: presentation only
INTRODUCTION
Contribution

• This work has the main goal of implementing a wide range of network functions (i.e. the Linux’s iptables interface) on top of a (previously designed) programmable stateful dataplane supported by an scalable accelerated SW and HW (FPGA based) implementations

• We built this work on top Open Packet Processor (OPP), a stateful dataplane that allows the implementation of per-flow Extended Finite State Machines (EFSM)

• Threefold contribution:
  – Implement a set of complex and commonly implemented network functions on top of a programmable datapath
  – Prove the flexibility of OPP, which in this work is the selected abstraction (and actual implementation) for deploying such VNFs
  – Provide an OPP software implementation that can dynamically scale to run on multiple CPU cores
Abstraction requirements

• Ideally, we need a fast path abstraction that simplifies the implementation of high-performance VNFs, while enabling a seamless adoption of future hardware accelerators

• Flexible **stateful** abstraction to model several network functions’ data plane while fulfilling the following requirements:
  – it is efficiently implementable in both software and hardware (i.e., smart NICs);
  – it enables the deployment of new functions at runtime;
  – it can support multiple concurrent functions

• Currently available abstractions (OpenFlow, P4, POF, Banzai, Domino) all (partially) fail the above mentioned requirements

• Chosen abstraction: **Open Packet Processor (OPP)**
OPEN PACKET PROCESSOR
Open Packet Processor

- In OPP *Extended Finite State Machines* (EFSM) are used to model stateful forwarding algorithms
- An EFSM is a finite state machine in which (1) state transitions depend also on a set of triggering conditions depending on data variables; (2) state transitions trigger the update of data variables
- OPP handles the states retrieval/update on a per flow basis
- It also allows *cross-flow state modification*
- The OPP machine model *extends the MATs pipeline model* assumed by OpenFlow
- The pipeline processes packets’ headers to define corresponding forwarding behaviors
- The OPP assumes packets headers are already parsed when passed to the pipeline, therefore, OPP can potentially leverage related work on programmable packet parsing and reconfigurable match tables
Open Packet Processor at a glance

Flow context retrieval
Tell me what flow the packet belongs to and what is its state
Condition verification

Does the flow context respect some (user defined) conditions?
Open Packet Processor at a glance

EFSM execution
Match current status and conditions and retrieve next state and update functions (+ apply packet actions)
Open Packet Processor at a glance

Context update

What is the flow I need to update?
(+ apply flow context update)
About OPP HW implementation

- OPP NetFPGA prototype has been already presented and no architectural/implementation enhancement is provided. Some practical insights:
  - 156.25 MHz clock, 64 bits data path from the Ethernet ports = 10gbps Ethernet ports (4 in this prototype)
  - OPP EFSM is implemented in small TCAMs (32 entries, 16 bits). The well known limited TCAM support on FPGA is not a real problem
  - *System parameters*: 4 flow registers, 8 conditions, 32 bit metadata, 8 global registers, 5 ALUs, 6 stages
  - *Flow context table*: (assuming) 4 stateful stages, 128 bit key, 4 32 bit flow registers → each stage holds 48k flow entries
- OPP FPGA prototype support the execution of the use cases defined in this work at line rate
THE IPTABLES USE CASE
iptables primer

• *iptables* is a well known Linux’s user interface to control the netfilter module

• With the *conntrack* module, netfilter supports a wide range of NFs:
  – filtering, NAT, stateful firewall, load balancer, anomaly detection ...

• Typical *iptables* command structure

```bash
iptables <command> <chain> <table> <match> <target>
iptables -A POSTROUTING -t nat -o eth0 -j MASQUERADE
```

• While in principle we support a great *majority* of the *iptables* interface, we focus only on three usecases (next slides)
iptables translation onto OPP

• iptables rules are translated into rules that configure a set of OPP stages
• For sake of clarity the OPP pipeline configured for this works supports the following:
  • MATCHES: all stateless header fields matched by OF1.3; state; statistics; limit; recent
  • TARGETS: DROP; ACCEPT; DNAT; SNAT; MASQUERADE, RECENT

The general structure of the OPP pipeline is depicted below

Note: for the use case in this work the limit and recent stages are ignored
USE CASE SCENARIOS
Use case network topology

Private LAN

Public DMZ

Access GW (Stateful FW)

DMZ: 8.0.0.0/24

LAN 10.0.0.0/24

The “outside world”
(e0 on subnet 1.0.0.0/8)
Use case 1: LAN/DMZ isolation

Packets between DMZ and LAN are forwarded only for flows initiated from the LAN.

In iptables terms, packets sent from the DMZ to the LAN in state “NEW” are dropped.

iptables -P FORWARD DROP
iptables -A FORWARD -i e2 -o e1 -j ACCEPT
iptables -A FORWARD -i e1 -o e2 -m state --state ESTABLISHED -j ACCEPT
Use case 2: load balancing with static NAT

Each first packet of a flow is “assigned” to 1 of the available servers of the cluster in the LAN.

The packet is DNATed toward S1, .., Sn accordingly.

All subsequent packets will be forwarded and DNATed toward the proper server.

```
iptables -t nat -A PREROUTING -i e0 -d 1.0.0.1 -p tcp --dport 80 -m statistic --mode nth --every 2 -j DNAT --to-destination 10.0.0.3:80

iptables -t nat -A PREROUTING -i e0 -d 1.0.0.1 -p tcp --dport 80 -m statistic --mode nth --every 1 -j DNAT --to-destination 10.0.0.2:80
```
Use case 3: Dynamic NAT

Each first packet of a flow is “assigned” to a new application port.

All packets belonging to the same flow will be NATed accordingly (and they will).

All response packets from “outside” will be “restored” to the original IP address and L4 port.

```
iptables -t nat -A POSTROUTING -i e2 -o e0 -j MASQUERADE
```
The big picture

**EFSM Table 0**
lookup/update = bflow

**EFSM Table 3**
lookup/update = (p.src, tcp.src)

**EFSM Table 4 (stateless)**
USE CASE 1: LAN/DMZ ISOLATION
Use case 1: forwarding allowed

EFSM Table 0 (stateful)
lookup/update = biflow

<table>
<thead>
<tr>
<th>state</th>
<th>inport</th>
<th>Dst_addr</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>SET_STATE(11, 20 s); SET_STATE(2, 20 s);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>GOTO 4; SET_STATE(2, 20 s); WRITE_METADATA(b0=1);</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>8.0.0.0/24</td>
<td>SET_STATE(12, 20 s); GOTO 4;</td>
</tr>
</tbody>
</table>

EFSM Table 4 (stateless)

<table>
<thead>
<tr>
<th>b0</th>
<th>inport</th>
<th>dest addr</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>DROP</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>OUT 2</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>8.0.0.0/24</td>
<td>OUT 1</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>10.0.0.0/24</td>
<td>OUT 2</td>
</tr>
</tbody>
</table>

**First packet (inport=2, state=0, dst=8.0.0.0/24)**

ACCEPT and set state = 12 (i.e. 1 pck received from port 2)
Use case 1: forwarding allowed

EFSM Table 0 (stateful)
lookup/update = biflow

<table>
<thead>
<tr>
<th>state</th>
<th>inport</th>
<th>Dst_addr</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>SET_STATE(11, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>8.0.0.0/24</td>
<td>SET_STATE(12, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>SET_STATE(11, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>8.0.0.0/24</td>
<td>SET_STATE(2, 20 s); WRITE_METADATA (b0=1); GOTO 4;</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>8.0.0.0/24</td>
<td>SET_STATE(12, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>SET_STATE(2, 20 s) WRITE_METADATA (b0=1); GOTO 4;</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>WRITE_METADATA (b0=1); GOTO 4;</td>
</tr>
</tbody>
</table>

EFSM Table 4 (stateless)

<table>
<thead>
<tr>
<th>b0</th>
<th>inport</th>
<th>dest addr</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>DROP</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>OUT 2</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>8.0.0.0/24</td>
<td>OUT 1</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>10.0.0.0/24</td>
<td>OUT 2</td>
</tr>
</tbody>
</table>

LAN  FW  DMZ

second packet, same flow (inport=1, state=12, dst=10.0.0.0/24)  
ACCEPT and set state to ESTABLISHED (2)
Use case 1: forwarding allowed

EFSM Table 0 (stateful)
lookup/update = biflow

<table>
<thead>
<tr>
<th>state</th>
<th>inport</th>
<th>Dst_addr</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>SET_STATE(11, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>8.0.0.0/24</td>
<td>SET_STATE(12, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>SET_STATE(11, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>8.0.0.0/24</td>
<td>SET_STATE(2, 20 s); WRITE_METADATA (b_0=1); GOTO 4;</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>8.0.0.0/24</td>
<td>SET_STATE(12, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>SET_STATE(2, 20 s) WRITE_METADATA (b_0=1); GOTO 4;</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>WRITE_METADATA (b_0=1); GOTO 4;</td>
</tr>
</tbody>
</table>

EFSM Table 4 (stateless)

<table>
<thead>
<tr>
<th>b0</th>
<th>inport</th>
<th>dest addr</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>DROP</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>OUT 2</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>8.0.0.0/24</td>
<td>OUT 1</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>10.0.0.0/24</td>
<td>OUT 2</td>
</tr>
</tbody>
</table>

LAN \[\x22\] FW \[\x22\] DMZ

all subsequent packets of the same flow (state=2)
**Use case 1:** forwarding denied

**EFSM Table 0 (stateful)**
lookup/update = biflow

<table>
<thead>
<tr>
<th>state</th>
<th>inport</th>
<th>Dst_addr</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>SET_STATE(11, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>8.0.0.0/24</td>
<td>SET_STATE(12, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>SET_STATE(11, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>8.0.0.0/24</td>
<td>SET_STATE(2, 20 s); WRITE_METADATA (b_0=1); GOTO 4;</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>8.0.0.0/24</td>
<td>SET_STATE(12, 20 s); GOTO 4;</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>SET_STATE(2, 20 s) WRITE_METADATA (b_0=1); GOTO 4;</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>WRITE_METADATA (b_0=1); GOTO 4;</td>
</tr>
</tbody>
</table>

**EFSM Table 4 (stateless)**

<table>
<thead>
<tr>
<th>b0</th>
<th>inport</th>
<th>dest addr</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>DROP</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10.0.0.0/24</td>
<td>OUT 2</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>8.0.0.0/24</td>
<td>OUT 1</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>10.0.0.0/24</td>
<td>OUT 2</td>
</tr>
</tbody>
</table>

**LAN**

**FW**

**DMZ**

**first packet (inport=1, state=0, dst=10.0.0.0/24)**

DROP and set state = 11 (i.e. 1 pck received from port 1)
SW ACCELERATION
OPP SW optimization

• **Starting point**: single core implementation based on ofsoftswitch13, an OpenFlow 1.3 compliant software switch very popular in the academic community, byt originaly designed without scalability concerns in mind...

• **Several code optimizations/redisign actions**:
  1. fast packet capture driver integration (PFQ)
  2. parallel (multi-pfq) computation
  3. zero-malloc optimization
  4. ofsoftswitch hashmap refactory
  5. zero copy
  6. batch processing
OFSoftSwitch performance and speedup factor
Stateless OFSoftSwitch: Acceleration contributions
Stateless OPP performance
Stateful OPP performance
Conclusions

• Flow-level states can be efficiently implemented in software
• Even complex network functions such implementation can be easily scaled to run on multi-core systems
• We remark that a subset of the OPP API (mainly OpenState) is currently under discussion for inclusion in OpenFlow v.1.6
• This work was partially supported by the European Commission in the frame of the BEBA project http://www.beba-project.eu