PN Lab 2

309552007 袁鈺勛

A. P4 Pipeline

a. Parser

```
state start {
    transition select(standard_metadata.ingress_port) {
        CPU_PORT: parse_packet_out;
        default: parse_ethernet;
    }
}

state parse_packet_out {
    packet.extract(hdr.packet_out);
    transition parse_ethernet;
}

state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition accept;
}
```

First, packet traverses the states in the parser. If it is from the controller, it will enter “parse_packet_out” state. If not, it will directly enter “parse_ethernet” state. After parsing, it will enter the ingress pipeline.

b. Ingress Pipeline

```
direct_counter(CounterType.packets) ether_counter;

action drop() {
    mark_to_drop(standard_metadata);
}

action send_to_controller() {
    standard_metadata.egress_spec = CPU_PORT;
}

action set_egress_port(egressSpec_t port) {
    standard_metadata.egress_spec = port;
}

table ethernet_forward {
    key = {
        hdr.ethernet.dst_addr          : ternary;
        hdr.ethernet.ether_type        : ternary;
        standard_metadata.ingress_port: ternary;
        hdr.ethernet.src_addr          : ternary;
    }
    actions = {
        drop;
        send_to_controller;
        set_egress_port;
        NoAction;
    }
    default_action = drop();
    size = 1024;
    counters = ether_counter;
}

apply {
    if (standard_metadata.ingress_port == CPU_PORT) {
        // Forward the packet in packet_out
        standard_metadata.egress_spec = hdr.packet_out.egress_port;
        hdr.packet_out.setInvalid();
    } else if (hdr.ethernet.isValid()) {
        ethernet_forward.apply();
    }
}
```

After parsing, packet will enter the ingress pipeline. Ingress pipeline first checks whether the packet is from the controller. If it is from the controller, ingress pipeline just sets the output port for it and set the "packet_out" header invalid. If not, it will traverse the "ethernet_forward" table to check whether there is a match. After checking the table, it will enter the egress pipeline.

c. Egress Pipeline

```
apply {
    if (standard_metadata.egress_port == CPU_PORT) {
        hdr.packet_in.setValid();
        hdr.packet_in.ingress_port = standard_metadata.ingress_port;
    }
}
```

After traversing ingress pipeline, packet will enter egress pipeline. If it will be sent to the controller, “packet_in” header will be set valid. Then, it will enter deparser.

d. Deparser

```
apply {
    packet.emit(hdr.packet_in);
    packet.emit(hdr.ethernet);
}
```

In the deparser, packet will be reconstructed. If it will be sent to the controller, a new “packet_in” header will be added to it.

B. Pipeconf App

There are two modules implemented in the pipeconf app. One is pipeliner, and the other is interpreter.

a. Pipeliner

```
if (obj.treatment() == null) {
    obj.context().ifPresent(c -> c.onError(obj, ObjectiveError.UNSUPPORTED));
}

final FlowRule.Builder ruleBuilder = DefaultFlowRule.builder() DefaultFlowRule.Builder
        .forTable(TABLE_ETHERNET_FORWARD) FlowRule.Builder
        .forDevice(deviceId)
        .withSelector(obj.selector())
        .fromApp(obj.appId())
        .withPriority(obj.priority())
        .withTreatment(obj.treatment());

if (obj.permanent()) {
    ruleBuilder.makePermanent();
} else {
    ruleBuilder.makeTemporary(obj.timeout());
}

switch (obj.op()) {
    case ADD:
        flowRuleService.applyFlowRules(ruleBuilder.build());
        break;
    case REMOVE:
```

```
        flowRuleService.removeFlowRules(ruleBuilder.build());
        break;
    default:
        log.warn("Unknown operation {}", obj.op());
}

obj.context().ifPresent(c -> c.onSuccess(obj));
```

Pipeliner will transform the forwarding objective from p4-learning-bridge app to flow rules.

b. Interpreter

```
if (!piTableId.toString().equals(TABLE_ETHERNET_FORWARD.toString()))
    throw new PiInterpreterException("Can map treatments only for 'ethernet_forward' table");

if (treatment.allInstructions().isEmpty()) {
    // 0 instructions means drop
    return PiAction.builder().withId(ACT_ID_DROP).build();
} else if (treatment.allInstructions().size() > 1) {
    // We understand treatments with only 1 instruction.
    throw new PiInterpreterException("Treatment has multiple instructions");
}

// Get the first and only instruction.
Instruction instruction = treatment.allInstructions().get(0);

if (instruction.type() == NOACTION)
    return PiAction.builder().withId(ACT_ID_NOP).build();

if (instruction.type() != OUTPUT) {
    // We can map only instructions of type OUTPUT.
    throw new PiInterpreterException(format("Instruction of type '%s' not supported", instruction.type()));
}

OutputInstruction outInstruction = (OutputInstruction) instruction;
PortNumber port = outInstruction.port();
```

```
if (!port.isLogical()) {
    // Forward the packet
    return PiAction.builder()
            .withId(ACT_ID_SET_EGRESS_PORT)
            .withParameter(new PiActionParam(ACT_PARAM_ID_PORT, port.toLong()))
            .build();
} else if (port.equals(CONTROLLER)) {
    // Send packet to controller
    return PiAction.builder()
            .withId(ACT_ID_SEND_TO_CONTROLLER)
            .build();
} else {
    throw new PiInterpreterException(format("Output on logical port '%s' not supported", port));
}
```

The images above are from the “mapTreatment” function in interpreter. The function translates the instruction in the treatment into Pi version such that P4 switch can understand.

```
TrafficTreatment treatment = packet.treatment();

// We support only packet-out with OUTPUT instructions.
List<OutputInstruction> outInstructions = treatment
        .allInstructions() List<Instruction>
        .stream() Stream<Instruction>
        .filter(i -> i.type().equals(OUTPUT))
        .map(i -> (OutputInstruction) i) Stream<Instructions.OutputInstruction>
        .collect(Collectors.toList());

if (treatment.allInstructions().size() != outInstructions.size()) {
    // There are other instructions that are not of type OUTPUT.
    throw new PiInterpreterException("Treatment not supported: " + treatment);
}

ImmutableList.Builder<PiPacketOperation> builder = ImmutableList.builder();
for (OutputInstruction outInst : outInstructions) {
    if (outInst.port().isLogical() && !outInst.port().equals(FLOOD)) {
        throw new PiInterpreterException(format(
                "Output on logical port '%s' not supported", outInst.port()));
    } else if (outInst.port().equals(FLOOD)) {
        // Create a packet operation for each switch port and prevent from flooding to input port
        long input_port = treatment.writeMetadata().metadata();
        final DeviceService deviceService = handler().get(DeviceService.class);
        for (Port port : deviceService.getPorts(packet.sendThrough())) {
            if (port.number().toLong() != input_port)
                builder.add(createPiPacketOp(packet.data(), port.number().toLong()));
        }
    } else {
        builder.add(createPiPacketOp(packet.data(), outInst.port().toLong()));
    }
}
return builder.build();
```

The images above are from the “mapOutboundPacket” function in interpreter. The function translates the output instructions into Pi version such that P4 switch can understand.

C. Workflow

```
// Setup flow-mod object
ForwardingObjective forwardingObjective = DefaultForwardingObjective.builder()
        .withSelector(selector)
        .withTreatment(treatment)
        .withPriority(DEFAULT_PRIORITY)
        .withFlag(ForwardingObjective.Flag.VERSATILE)
        .fromApp(app_id)
        .makeTemporary(DEFAULT_TIMEOUT)
        .add();

// Forward flow-mod object
flowObjectiveService.forward(context.inPacket().receivedFrom().deviceId(), forwardingObjective);
```

First, forwarding objective will be contructed in the p4-learning-bridge app.

```
final FlowRule.Builder ruleBuilder = DefaultFlowRule.builder() DefaultFlowRule.Builder
        .forTable(TABLE_ETHERNET_FORWARD) FlowRule.Builder
        .forDevice(deviceId)
        .withSelector(obj.selector())
        .fromApp(obj.appId())
        .withPriority(obj.priority())
        .withTreatment(obj.treatment());

if (obj.permanent()) {
    ruleBuilder.makePermanent();
} else {
    ruleBuilder.makeTemporary(obj.timeout());
}
```

Secondly, forwarding objective will be decomposed into flow rules in pipeliner.

```
if (!port.isLogical()) {
    // Forward the packet
    return PiAction.builder()
            .withId(ACT_ID_SET_EGRESS_PORT)
            .withParameter(new PiActionParam(ACT_PARAM_ID_PORT, port.toLong()))
            .build();
} else if (port.equals(CONTROLLER)) {
    // Send packet to controller
    return PiAction.builder()
            .withId(ACT_ID_SEND_TO_CONTROLLER)
            .build();
```

Then, the treatment in the flow rules will be translated into Pi version. Last, these Pi-version flow rules will be installed into the p4 switch.

D. Flooding Function

```
private void flood(PacketContext context, PortNumber input_port) {
    if (topologyService.isBroadcastPoint(topologyService.currentTopology(), context.inPacket().receivedFrom()))
        packet_out(context, PortNumber.FLOOD, input_port);
    else
        context.block();
}
```

There is a flooding function in the p4-learning-bridge app. Packet will be sent to the port with number “FLOOD”.

```
} else if (outInst.port().equals(FLOOD)) {
    // Create a packet operation for each switch port and prevent from flooding to input port
    long input_port = treatment.writeMetadata().metadata();
    final DeviceService deviceService = handler().get(DeviceService.class);
    for (Port port : deviceService.getPorts(packet.sendThrough())) {
        if (port.number().toLong() != input_port)
            builder.add(createPiPacketOp(packet.data(), port.number().toLong()));
    }
```

In the interpreter, packet will be sent to all ports in the switch except the port from which the packet came. Interpreter will create a Pi packet operation for each port to flood the packet.