Smashing OpenFlow’s “atomic” actions: programmable data plane packet manipulation in hardware

Salvatore Pontarelli, Marco Bonola, Giuseppe Bianchi
CNIT / University of Roma Tor Vergata,
Email: {name.surname}@uniroma2.it
Outline

• Introduction

• Programmable SDN action

• PMP (Packet Manipulation Processor) Architecture

• Example of PMP applications

• Conclusions
The SDN paradigm

Traditional networking

Software-Defined Networking

smart, slow, (logically) centralized

Programmable switch

API to the data plane (e.g., OpenFlow)

dumb, fast
OpenFlow proposes an abstract model of a programmable flow table which permits the device programmer to broadly specify a flow via an header matching rule, associate forwarding/processing actions to the matching packets, and access statistics associated to the specified flow.
Action Programmability

• Actions in OpenFlow have mainly remained “atomic” and to the best of our knowledge very little work has addressed their flexibility.
• The programmer can only “select” which action should be associated to the outcome of a match, being such selection restricted to a set of actions (e.g., drop, output to port, push/pop VLAN/MPLS tag, etc) preimplemented in the device by the vendor.

Action programmability can:

• Allowing SDN switches to run complex network task such as NAT, generic encapsulation etc.
• Help the introduction/deployment of new/enhanced network protocols (new encapsulation methods, new/experimental protocol fields, etc.)
• Adding new functionality such as programmable packet generation or in-band network measurements and debug.
SDN action classes:

We identify three classes of actions starting from the set of pre-implemented actions provided by OpenFlow (and also by the newcomers SDN languages such as P4):

1. Header fields actions: these actions will add/modify/remove one of the header fields of the packet that is processed

2. Packet level actions: these actions acts at packet level performing drop(), clone(), recirculate() actions

3. Packet generation actions: while not directly available in current SDN, we believe that this will be a feature of future SDN platforms.
Header fields actions

(a) the packet before and after the add field operation

the packet stored in a 32 bits memory before and after the add field operation
Dispatching actions

These operations are a set of read/write from/to the queue memory in which the packet is stored, to another location (an I/O memory mapped location (an output port) or another location inside the buffer memory (e.g. to implement the P4 clone() operation).

- the bandwidth of the memory read/write operation is directly related to the memory data-width
- the use of specific instructions that move data from one memory address to another memory address can increase the throughput.
- Memory can be shared between switch queue and PMP
Packet generation actions

The actual packet to send outside the switch is forged copying the data coming from the template, with suitable modification of some packet headers/data.
The current SDN architectures are based on a pipeline of MAT (match-action table) stages forming an ingress and an egress pipeline.

The PMP can be inserted at the end of the egress pipeline (it is the mirror block of the ingress parser)
PMP architecture

- RISC-like structure
- Plain memory structure (no cache)
- Multiple memory accesses
### PMP instruction set

- minimal ISA
- loop instructions (DSP-like)
- directly I/O mov instructions
- multiple size instructions (B/H/W/D)

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Instructions</th>
<th>Memory mode</th>
<th>Operands</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic ALU Operations</td>
<td>NOP, AND, OR, XOR,</td>
<td>register-register</td>
<td>rd,rs1,rs2,</td>
<td>standard logic operations</td>
</tr>
<tr>
<td></td>
<td>XNOR, NOT</td>
<td></td>
<td>rd,rs1,imm</td>
<td></td>
</tr>
<tr>
<td>Arithmetic ALU</td>
<td>ADD, ADC, SUB, SBC, MUL</td>
<td>register-register</td>
<td>rd,rs1,rs2</td>
<td>standard arithmetic operations</td>
</tr>
<tr>
<td>Operations</td>
<td></td>
<td></td>
<td>rd,rs1,imm</td>
<td></td>
</tr>
<tr>
<td>Shift/Rotate Operations</td>
<td>LSL (Logical Shift</td>
<td>register-register</td>
<td>rd,rs1,rs2</td>
<td>performs logic and arithmetic</td>
</tr>
<tr>
<td></td>
<td>Left)</td>
<td></td>
<td>rd,rs1,imm</td>
<td>shift/rotate operations</td>
</tr>
<tr>
<td></td>
<td>LSR (Logical Shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Right)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ASR (Arithmetic Shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Right)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROR (Rotate Right)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control flow</td>
<td>B(cond) (Branch with</td>
<td>register-register</td>
<td>imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>condition)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BL (Branch and Link</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with condition)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RET (Return), HALT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/Store</td>
<td>ldb, ldh, ldw</td>
<td>memory-register</td>
<td>rd, imm</td>
<td>the operations move 8, 16 or 32 bits</td>
</tr>
<tr>
<td></td>
<td>stb, sth, stw</td>
<td>memory-register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>movb, movh, movw,</td>
<td>memory-memory</td>
<td>rd,addr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movd, movq</td>
<td></td>
<td>moves move up</td>
<td>128 bits from [addr] to [rd]</td>
</tr>
<tr>
<td>out</td>
<td>outb, outh, outw, outd, outq</td>
<td>memory-port</td>
<td>rd, addr</td>
<td>outs move from [addr] to the rd output port.</td>
</tr>
<tr>
<td>memory data movement</td>
<td>meml (Memory loop)</td>
<td>memory-memory</td>
<td>rd,rs1,rs2</td>
<td></td>
</tr>
<tr>
<td>memory data movement</td>
<td></td>
<td></td>
<td>moves rs2 bytes</td>
<td></td>
</tr>
<tr>
<td>output data movement</td>
<td>oult (output loop)</td>
<td>memory-port</td>
<td>rd,rs1,rs2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moves rs2 bytes</td>
<td></td>
</tr>
</tbody>
</table>
unaligned PMP memory

- combined PMP memory / packet queue
- multiple 8 bits banks for unaligned access
- dual port for PMP/switch interface

DP RAM

Dout[7:0]  Din[7:0]

ADDR_A
ADDR_B

INTERCONN MATRIX

DP RAM

Dout[7:0]  Din[7:0]

ADDR_A
ADDR_B

DP RAM

Dout[7:0]  Din[7:0]

ADDR_A
ADDR_B

controL LOGIC

RD_ADDR
WR_ADDR

DATA_IN[127:0]  DATA_OUT[127:0]
Developed applications

• Network Address and Port Translation
  The throughput achievable goes from 11.6 Gb/s (worst case) to 90 Gb/s (max size packets)

• ARP reply generation
  The ARPreply code is always executed in 18 clock cycles, which correspond to a throughput for this application of around 28.4 Gb/s.

• IPinIP encapsulation
  (harder than other encapsulation types: TTL, IP fragmentation etc). The throughput achievable goes from 12.2 Gb/s (worst case) to 90 Gb/s (max size packets)
Conclusions

• We proposed an architecture to adding action programmability to SDN platforms
• We selected a specific instruction set tailored for packet manipulation
• We designed a CPU architecture for packet manipulation able to provide multiple 10 Gb/s of throughput.
• We tested the proposed CPU with several network application to test the effectiveness of the PMP