Data Plane Programmability the next step in SDN

Giuseppe Bianchi
CNIT / University of Roma Tor Vergata

Credits to: *M. Bonola, A. Capone, C. Cascone, S. Pontarelli, D. Sanvito, M. Spaziani Brunella, V. Bruschi*

EU Support: SUPERFLUIDITY, Beba BEhavioural BAsed forwarding

The SDN/OpenFlow Model

- **Very elegant and performing**
  - Switch as a «sort of» programmable device
  - Line-rate/fast-path (HW) performance
  - Can be «repurposed» as switch, router, firewall, etc
- **...but...**
  - Static rules
  - All intelligence in controller
  - Lack of flexibility and espessivity: more of a config than a program!

OpenFlow's platform agnostic «program»: (abstract) Flow table

Match 1 → Act A
Match 2 → Act B

Controller

Run-time deployment (flow-mod)

Match primitives

Match 1 → Act A
Match 2 → Act B

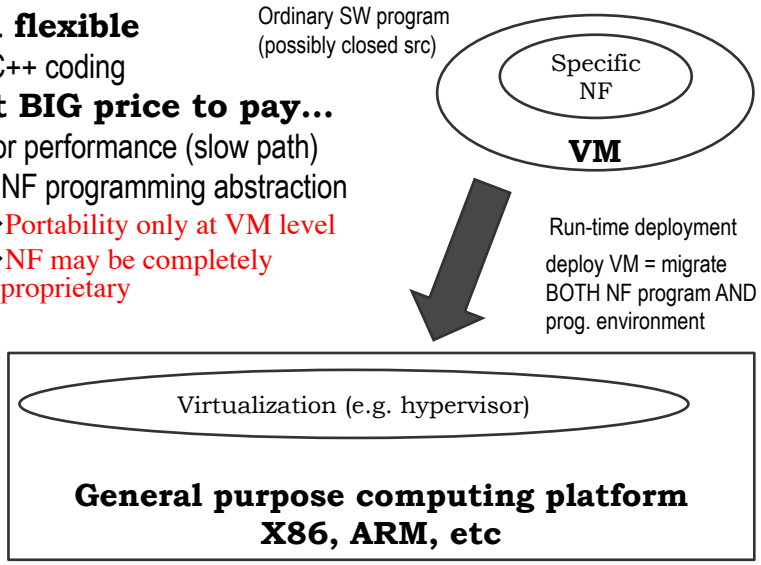
Pre-implemented actions

Networking-specific programmable device
OpenFlow (HW/SW) switch

The NVF model (opposite extreme)

➔**Ultra flexible**
- ⇨C/C++ coding

➔**...but BIG price to pay...**
- ⇨Poor performance (slow path)
- ⇨No NF programming abstraction
  - ➔Portability only at VM level
  - ➔NF may be completely proprietary



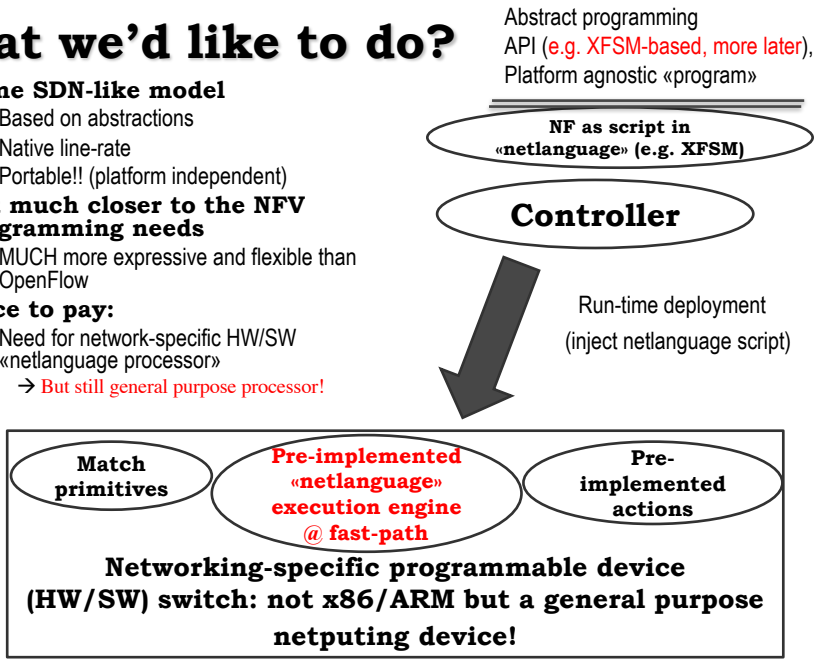
Giuseppe Bianchi

What we'd like to do?

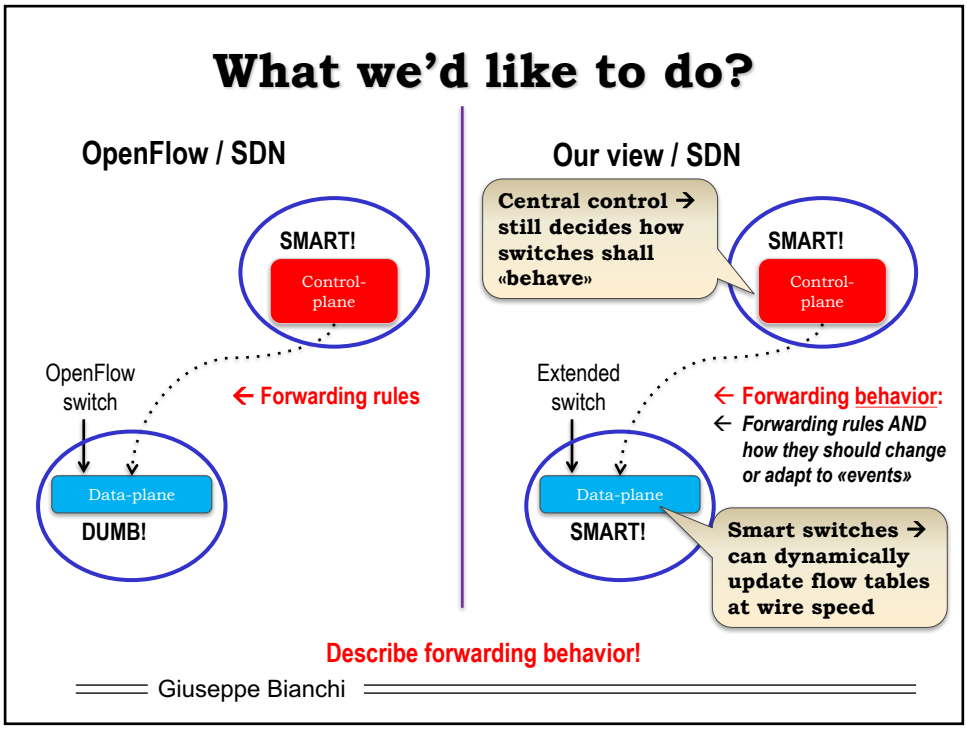
➔ **Same SDN-like model**
- ⇨ Based on abstractions
- ⇨ Native line-rate
- ⇨ Portable!! (platform independent)

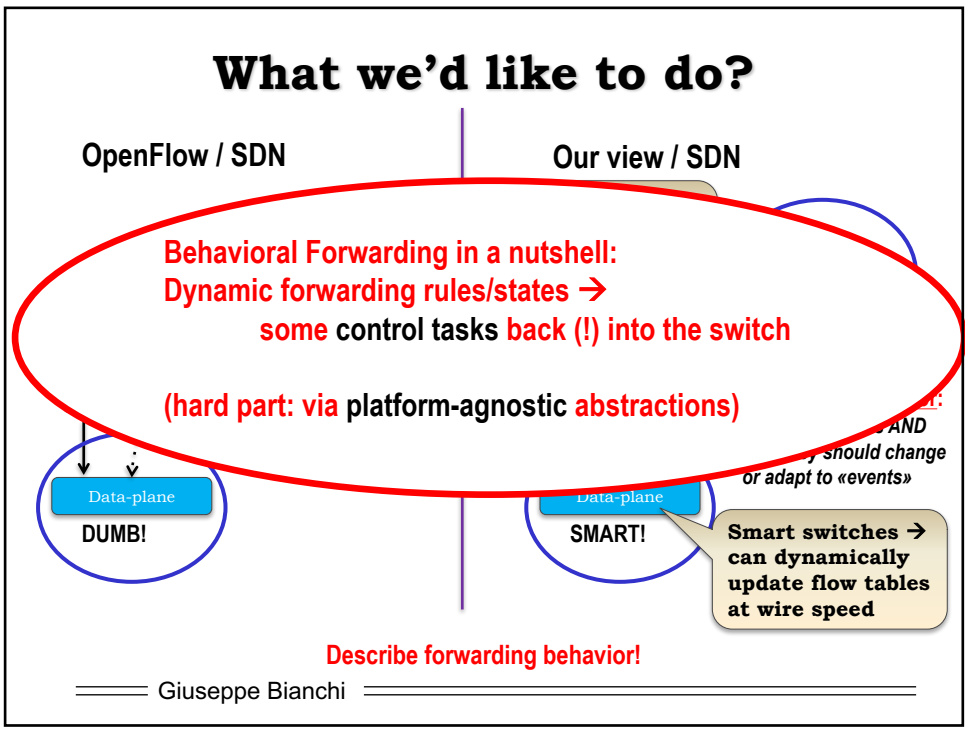
➔ **But much closer to the NFV programming needs**
- ⇨ MUCH more expressive and flexible than OpenFlow

➔ **Price to pay:**
- ⇨ Need for network-specific HW/SW «netlanguage processor»
  - ➔ But still general purpose processor!



Giuseppe Bianchi


What we'd like to do?
OpenFlow / SDN
Our view / SDN
SMART!
Control-plane
OpenFlow switch
← Forwarding rules
Data-plane
DUMB!
Central control → still decides how switches shall «behave»
SMART!
Control-plane
Extended switch
← Forwarding behavior:
← Forwarding rules AND how they should change or adapt to «events»
Data-plane
SMART!
Smart switches → can dynamically update flow tables at wire speed
Describe forwarding behavior!
Giuseppe Bianchi


What we'd like to do?
OpenFlow / SDN
Our view / SDN
Behavioral Forwarding in a nutshell:
Dynamic forwarding rules/states →
some control tasks back (!) into the switch
(hard part: via platform-agnostic abstractions)
Data-plane
DUMB!
Data-plane
SMART!
Smart switches → can dynamically update flow tables at wire speed
Describe forwarding behavior!
Giuseppe Bianchi

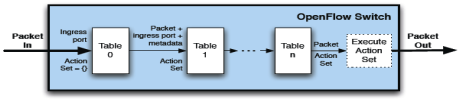
Towards data plane programmability: state of the art

Giuseppe Bianchi

OpenFlow evolutions

➔ **Pipelined tables from v1.1**

- ⇨ Overcomes TCAM size limitation
- ⇨ Multiple matches natural



➔Ingress/egress, ACL, sequential L2/L3 match, etc.

➔ **Extension of matching capapilities**

- ⇨ More header fields
- ⇨ POF (Huawei, 2013): complete matching flexibility!

➔ **Openflow «patches» for (very!) specific processing needs and states**

- ⇨ Group tables, meters, synchronized tables, bundles, typed tables (sic!), etc
- ⇨ Not nearly clean, hardly a «first principle» design strategy
- ⇨ A sign of OpenFlow structural limitations?

Giuseppe Bianchi

Programming the data plane: The P4 initiative (2014)

- **SIGCOMM CCR 2014. Bosshart, McKeown, et al. P4: Programming protocol-independent packet processors**
  - Dramatic flexibility improvements in packet processing pipeline
    - **Configurable packet parser** → parse graph
    - **Target platform independence** → compiler maps onto switch details
    - **Reconfigurability** → change match/process fields during pipeline
- **Feasible with HW advances**
  - Reconfigurable Match Tables, SIGCOMM 2013
  - Intel's FlexPipe™ architectures
- **P4.org: Languages and compilers**
  - Further support for «registry arrays» and counters meant to persist across multiple packets
    - Though no HW details, yet

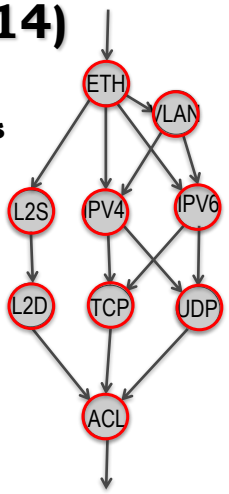


Table Graph

Programming the data plane: The P4 initiative (2014)

OpenFlow 2.0 proposal?

Stateful processing, but only «inside» a packet processing pipeline!

Not yet (clear) support for stateful processing «across» subsequent packets in the flow

"[…] extend P4 to express stateful processing",
Nick McKeown talking about P4 @ OVSconf Nov 7, 2016

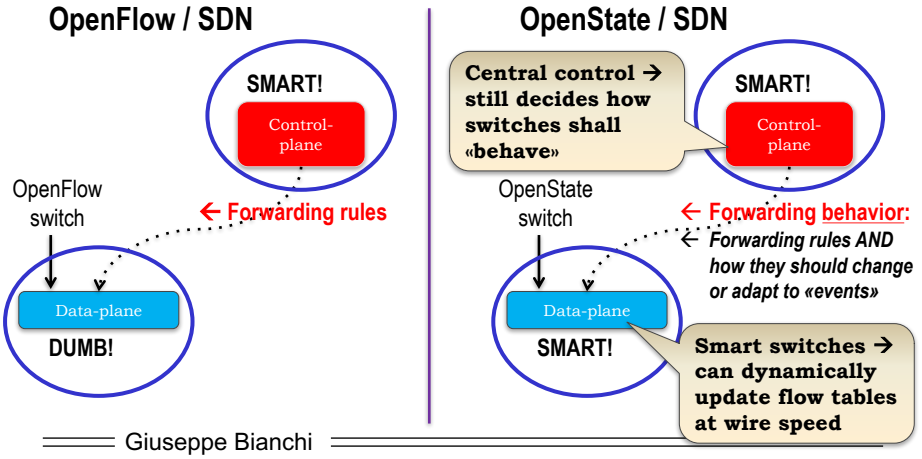
Table Graph

Giuseppe Bianchi

OpenState, 2014

➔ **Our group, SIGCOMM CCR 2014; surprising finding: an OpenFlow switch can «already» support stateful evolution of the forwarding rules**

⇨ With almost marginal (!) architecture modification



Our findings at a glance

➔**Any control program that can be described by a Mealy (Finite State) Machine is already (!) compliant with OF1.3**

➔**MM + Bidirectional flow state handling requires minimal hardware extensions to OF1.1+**

Details in G. Bianchi, M. Bonola, A. Capone, C. Cascone,
"**OpenState: programming platform-independent stateful OpenFlow applications inside the switch**", ACM SIGCOMM Computer Communication Review, vol. 44, no. 2, April 2014.

Our findings at a glance

→Any co… …n be de… …)

Candidate for inclusion in (as early as!) OpenFlow 1.6

Ongoing discussion in ONF

→ very concrete, fine tuning of details

Pragmatism and compatibility with OpenFlow→ key asset for being considered

De…

"OpenSta…

OpenFlow applica… …mputer Communication Review, vol. 44, no. 2, April 2014.

Remember OF match/action API

Programmabile logic

Vendor-implemented

Matching Rule	Action

1. FORWARD TO PORT
2. ENCAPSULATE&FORWARD
3. DROP
4. …

Extensible

Pre-implemented matching engine

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport

What is the OF abstraction, formally?

➔ **Packet header match = "*Input Symbol*" in a finite set $I=\{i_1, i_2, \ldots, i_M\}$.**
- ⇨ One input symbol = any possible header match
- ⇨ Possible matches pre-implemented; cardinality depends on match implementation
- ⇨ Theoretically, it is irrelevant how the Input Symbols' set ***I*** is established
  - →i.e. each input symbol = Cartesian combination of multiple header field matches, further including "wildcard" matches;
  - →E.s. incoming packet destination port = 5238 AND source IP address is 160.80.82.1, and the VLAN tag is 1111, etc.

➔ **OpenFlow actions = "*Output Symbols*" in finite set $O=\{o_1, o_2, \ldots, o_N\}$**
- ⇨ Pre-implemented actions

➔ **OpenFlow's match/action abstraction: a map $T : I \rightarrow O$**
- ⇨ all what the third party programmer can specify!

Giuseppe Bianchi

Reinterpreting (and extending) the OpenFlow abstraction

➔**OpenFlow map is trivially recognized to be a very special and trivial case of a Mealy Finite State Machine**

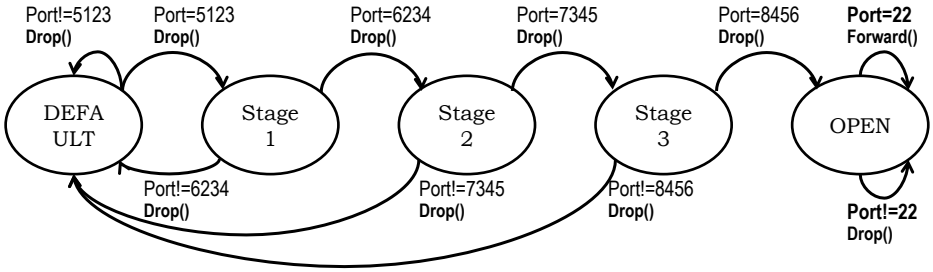
➔**T : {default-state}× I →{default-state}× O,**

➔**i.e. a Finite State Machine with output, where we only have one single (default) state!**

➔**By adding (per-packet) retrieval and update of states, OpenFlow can be turned it into a Mealy machine executor!!**

Giuseppe Bianchi

If an application can be «abstracted» in terms of a mealy Machine…



Example: Port Knocking firewall
knock «code»: 5123, 6234, 7345, 8456 → then open Port 22

… it can be transformed in a Flow Table!

Ipsrc: ??

State DB

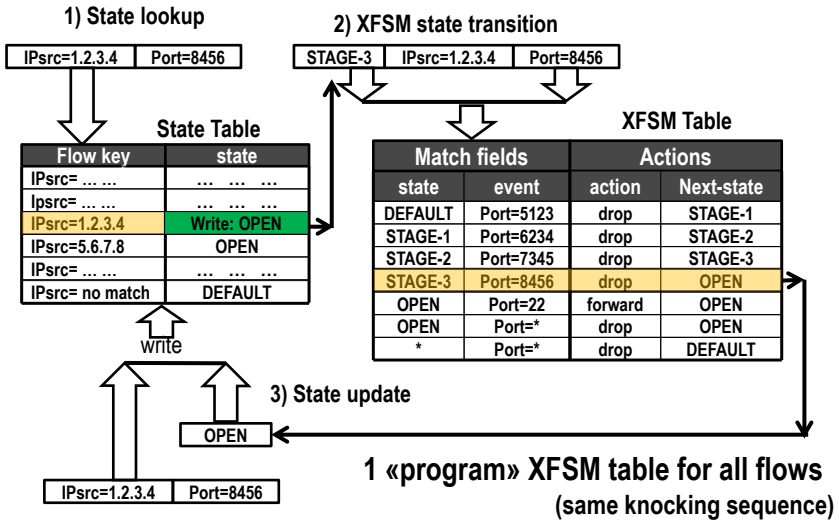
MATCH: <state, port> → ACTION: <drop/forward, state_transition>
Plus a state lookup/update

Match fields		Actions	
state	event	action	Next-state
DEFAULT	Port=5123	drop	STAGE-1
STAGE-1	Port=6234	drop	STAGE-2
STAGE-2	Port=7345	drop	STAGE-3
STAGE-3	Port=8456	drop	OPEN
OPEN	Port=22	forward	OPEN
OPEN	Port=*	drop	OPEN
*	Port=*	drop	DEFAULT

Ipsrc→OPEN

State DB

Putting all together



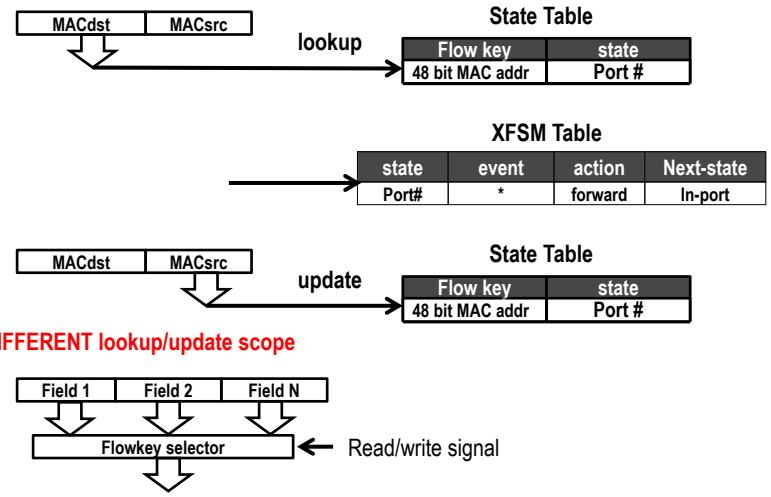
1 «program» XFSM table for all flows

(same knocking sequence)

N states, one per (active) flow

Cross-flow state handling

➔ **Yes but what about MAC learning, multi-port protocols (e.g., FTP), bidirectional flow handling, etc?**



towards 'true' data plane programmability

OpenState → Open Packet Processor?

- ArXiv: G. Bianchi, S. Pontarelli, M. Bonola, A. Capone, C. Cascone, D. Sanvito, "**Open Packet Processor**"

Giuseppe Bianchi

Mealy Machine: nice but insufficient!

→«true» Flow processing requires memory, registries, counters, etc

⇨State alone is insufficient

→«true» flow processing requires operations (compare, add, shift, etc)

⇨OpenFlow (forwarding) actions are insufficient

→«true» flow processing requires… «processing»

⇨Processing = CPU: cannot afford any ordinary CPUs at ns time scales wire speed!

Can we further evolve OpenState into an architecture equivalent to a "full" CPU (Without using any CPU?)

AND CAPABLE OF EXECUTING A PLATFORM AGNOSTIC ABSTRACTION?

Giuseppe Bianchi

Trivial example: state alone inefficient

- **Different forwarding for long vs short flows**
  - E.g. Long flows = packet count >=5

state	event	action	Next-state
DEFAULT	*	Fwd A	1 PKT
1 PKT	*	Fwd A	2 PKT
2 PKT	*	Fwd A	3 PKT
3 PKT	*	Fwd A	4 PKT
4 PKT	*	Fwd A	LONG
LONG	*	Fwd B	LONG

- **Better approach:**
  State + register (pkt count) + condition
  - saving TCAM entries

state	event	condition	action	Next-state	update
DEFAULT	*	*	Fwd A	SHORT	R=1
SHORT	*	R<5	Fwd A	SHORT	R++
SHORT	*	R>=5	Fwd B	LONG	-
LONG	*	*	Fwd B	LONG	-

Trivial example: state alone insufficient

➔Drop (or mark) traffic flow whose rate «suddenly» increases

⇨How to compute rate? (must perform arithmetic operation)

⇨State changes (e.g. green, yellow, red) not triggered by packet header fields or packet arrivals, but by conditions on rates

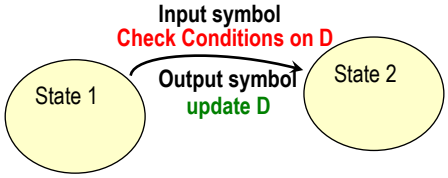
⇨No way to cast into a mealy machine!

Idea: from Mealy machines to XFSM

- ***Extended Finite State Machines*** (XFSM):
  finite state machines in which
  - System stores state labels AND variables;
  - state transitions depends also on a set of triggering conditions depending on data variables;
  - state transitions trigger the update of data variables

Extended finite state machines: much more general!

➔ **Mealy Machines: 4-tuple**
- ⇨ ***I, O, S***
- ⇨ ***T*:S×*I*→S×*O***



➔ **XFSM: 7-tuple**
- ⇨ ***I, O, S (Input symbols, output symbols, states)***
  - →As before, S = User-defined
- ⇨ ***D*=D_1×…×D_n** n-dimensional linear space
  - →Registries!!! Global or (user-defined) per flow!!
- ⇨ ***F* =** set of enabling functions f_i:D→{0,1}
  - →Boolean Conditions on registries!!!
- ⇨ ***U* =** set of update functions u_i:D→D
  - →Update of the registry values!
- ⇨ ***T*:S×*I*×F→S×*O*×U** the actual XFSM transition
  - →A map→ can be implemented by the TCAM!

Evolution of the abstractions

OpenFlow:

map

$$T : I \rightarrow O$$

I: match fields

O: actions

OpenState:

Mealy State Machine

$$T : S \times I \rightarrow S \times O$$

S: State

Open Packet Processor:

Extended State Machine

$$T : S \times F \times I \rightarrow S \times U \times O$$

D: Registers

$F: D \rightarrow \{0,1\}$ Conditions

$U: D \rightarrow D$ Update functions

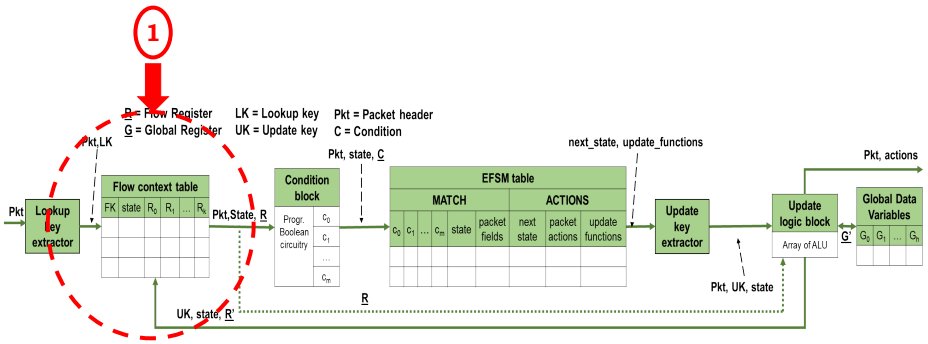
Towards an Open Packet Processor

➔HW architecture «executing» an XFSM

➔OpenState basic architecture + three major improvements

1. State DB → state + flow registers DB
2. Condition logic block → evaluates conditions
3. Update logic block → performs update operations

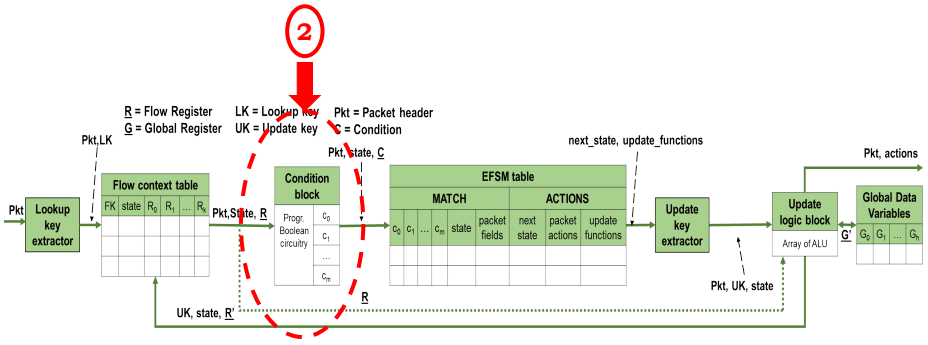
Open Packet Processor at a glance



Flow context retrieval

Tell me what flow the packet belongs to
and what is its state (and associated registries)

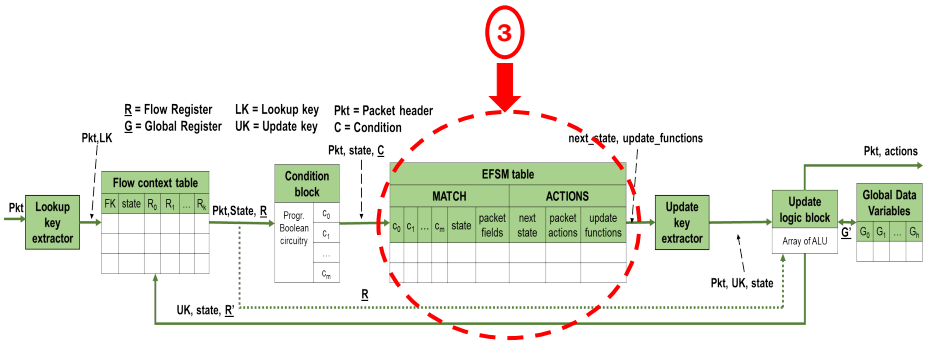
Open Packet Processor at a glance



Condition verification

Does the flow context respect
some (user defined) conditions?

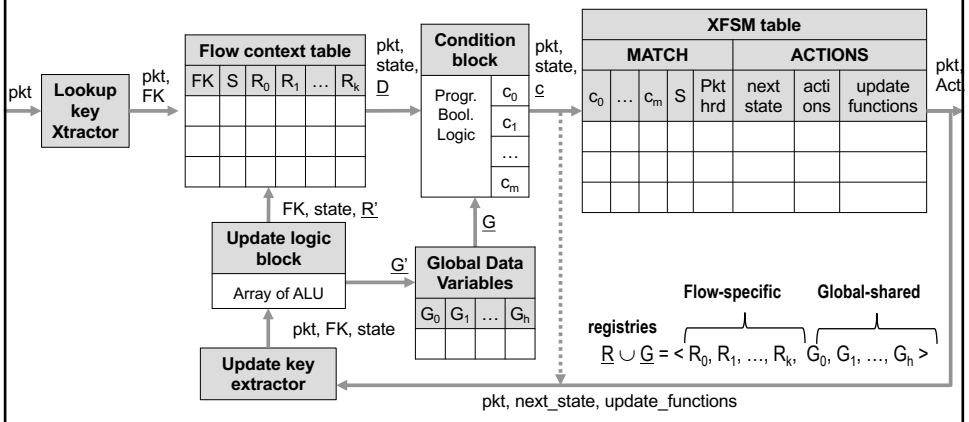
Open Packet Processor at a glance



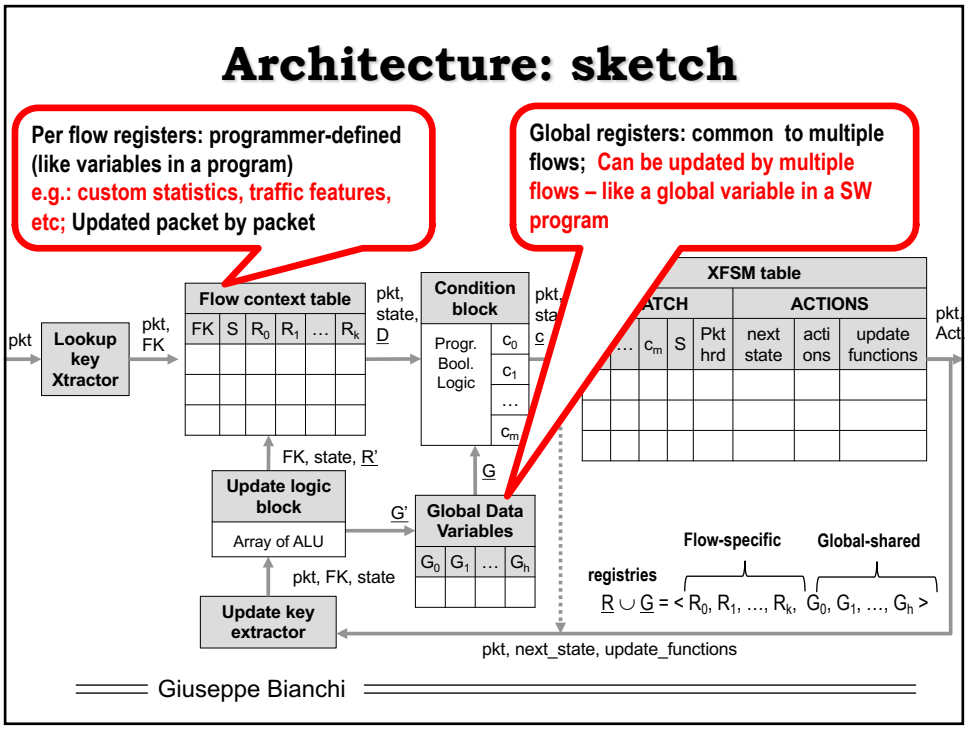
XFSM execution

Match current status and conditions and retrieve next state and update functions (+ apply packet actions)

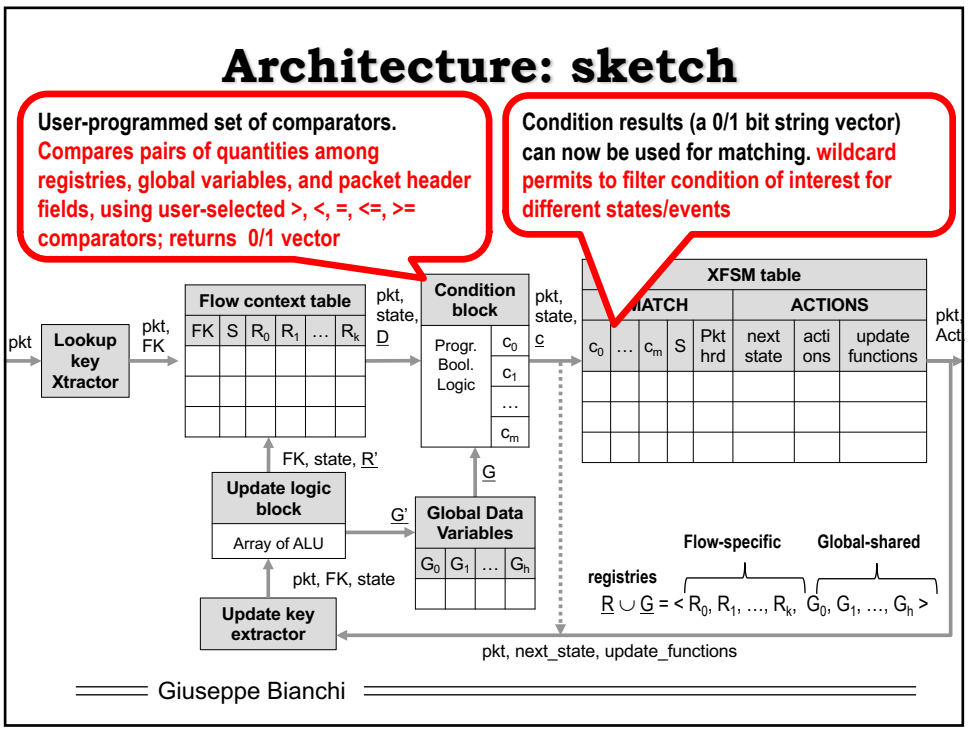
Architecture: sketch



Architecture: sketch

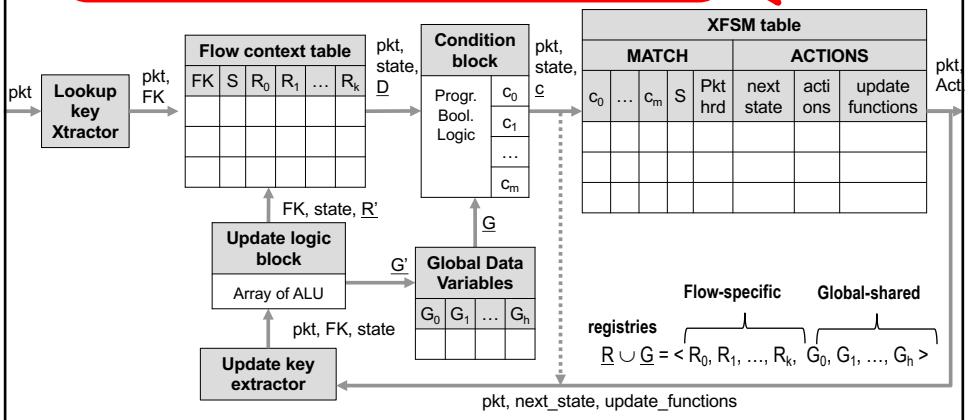


Architecture: sketch



Architecture: sketch

Match engine: standard TCAM!
Performs the XFSM transition
T: state x events x conditions → state x actions x update_fcts
i.e. the state machine "execution"!
(TCAM is used as "the" processor/CPU!)

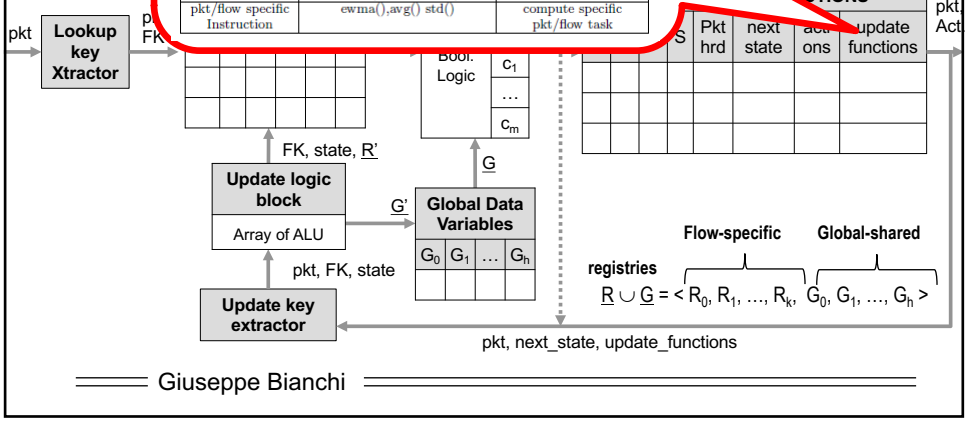


Giuseppe Bianchi

Architecture: sketch

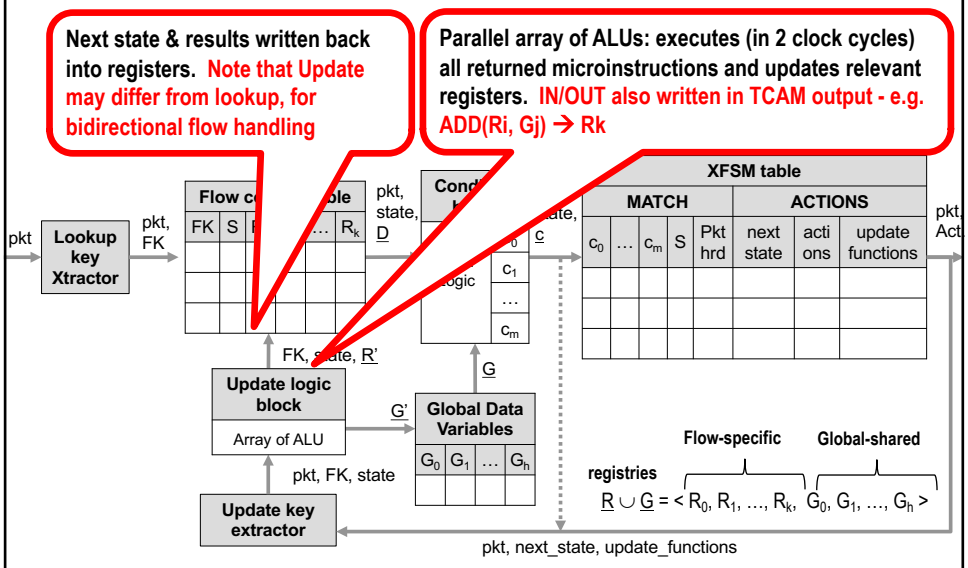
Returns microinstructions (of a domain-specific custom ALU instruction set) to be applied

Instruction Type	Instructions	note
Logic ALU Instruction	NOP, AND, OR, XOR, NOT	standard logic operations
Arithmetic ALU Instruction	ADD,ADC, SUB,SBC,MUL	standard arithmetic operations
Shift/Rotate Instruction	LSL (Logical Shift Left) LSR (Logical Shift Right) ASR (Arithmetic Shift Right) ROR (Rotate Right)	performs logic and arithmetic shift/rotate operations
pkt/flow specific Instruction	ewma(),avg() std()	compute specific pkt/flow task



Giuseppe Bianchi

Architecture: sketch



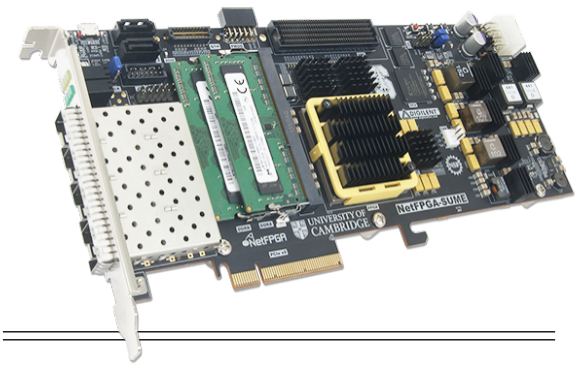
NetFPGA prototype

➔ **HW proof of concept implementation of OPP**

➔ **Target development board: NetFPGA SUME**

⇨ express PCI x8 mother card equipped with the XILINX Virtex7 FPGA

➔ **The prototype serves to gather insights on the HW limitation, design challenges and overall feasibility**



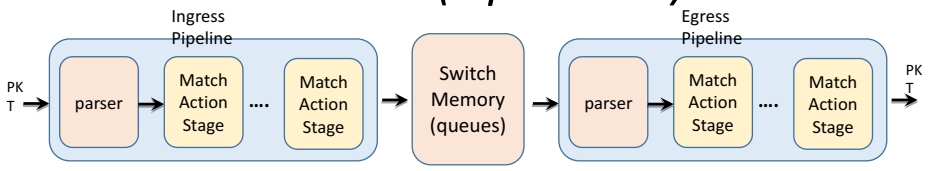
FPGA Prototype details

- A NetFPGA SUME can currently host up to 6 stateful OPP Stages.
  Each OPP stage is composed by:
  - 5 ALUs
  - 8 Global registers
  - 8 conditions
  - 4 per-flow registers for each entry
  - 32K entries of 288 bits (128 bits for the key +128 bits flow registers +32 bits state label)
  - 1 32x160 bits TCAM
- FPGA resources:
  - # Slice LUTs 22276 (5%)
  - # Block RAMs: 194 (13%)
- Last generation FPGA could provide 10x memory resources
- 156.25 MHz clock, 64 bits data path from the Ethernet ports = 10gbps Ethernet ports (4 in this prototype)

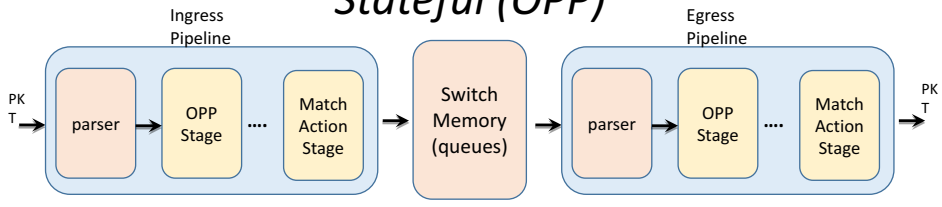
Giuseppe Bianchi

Switch pipeline

Stateless (OpenFlow)

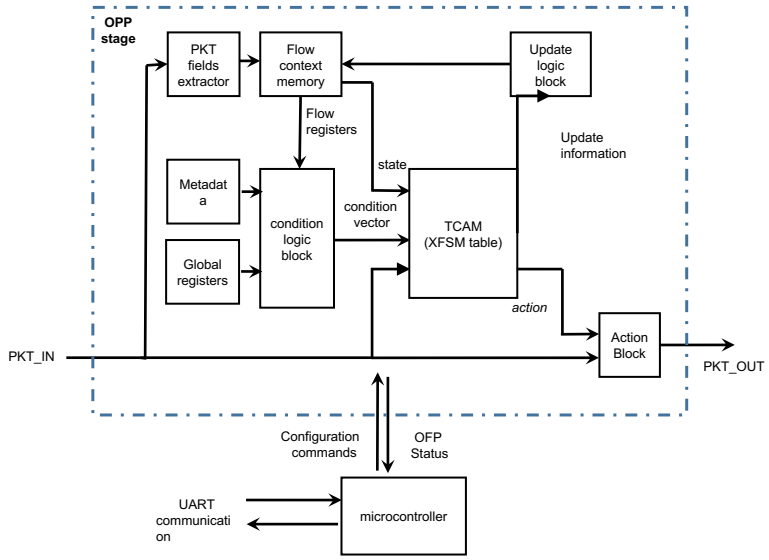


Stateful (OPP)

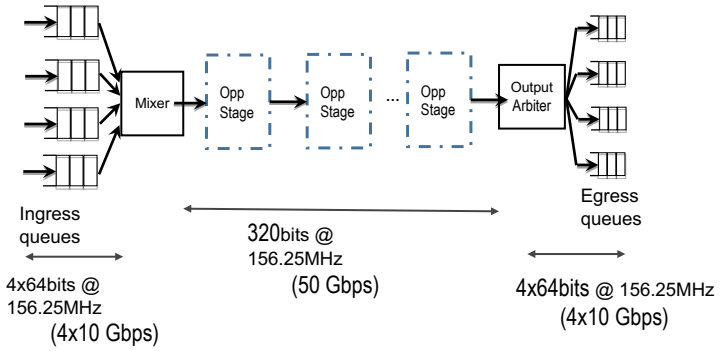


Giuseppe Bianchi

OPP stage



OPP data path



Capacity (FPGA and predicted ASIC)

FPGA not optimized (esp. TCAM)

- Throughput :
  - 40 Gbps on FPGA @156MHz
  - **640 Gbps on ASIC @1GHz**
- Number of flows in hash table:
  - 4K on FPGA,
  - **up to 2M on ASIC**
- Number of flows in TCAM:
  - 32x160b on FPGA (X 6)
  - **up to 256K on ASIC**

A TCAM-based packet processing engine!

➔ **Extreme flexibility!**

⇨ XFSM 'programs' almost flexible as ordinary programming language

→can define variables, store and change values, compute features, etc

➔ **Guaranteed wire speed!**

⇨ Fixed time per-packet computational loop

→6 clock cycles in our ongoing HW design

➔ **Ongoing work:**

⇨ Complete design, understand and overcome limitations, exploit it for more advanced use cases

⇨ Use XFSMs as 'machine code' for higher level language → compilation

➔ **(currently two tech limitations)**

⇨ Only 1 ALU operation per each packet → pipelined ALU arrays possible, but would increase processing time and yield more complex configuration

⇨ ALUs only in update, not in conditions → does not permit conditions such as (R1+R2>100)

→Solution (not nice, but workaround): compute R1+R2 → R3 during previous packet, then use (R3>100)

Layman OPP example: intra-flow stateful load balancing

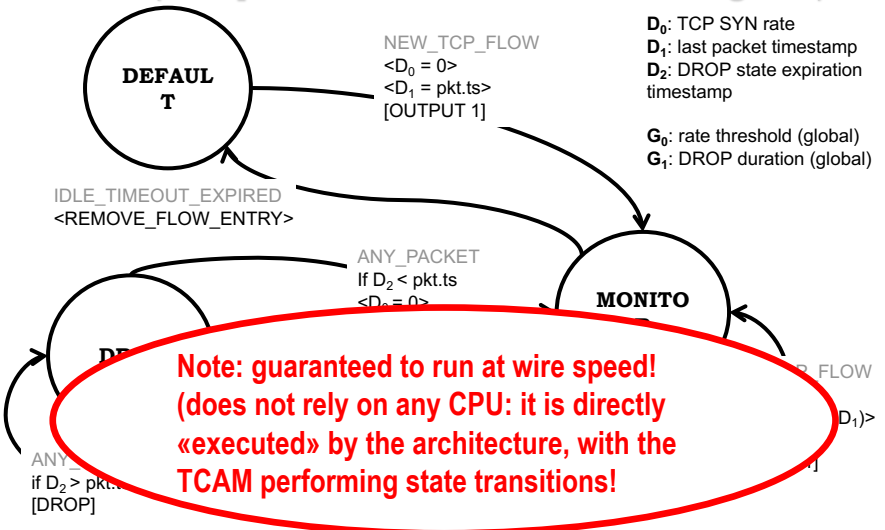
- **If next packet arrives before given deadline DELTA, stick to current path, otherwise (may) pick a new path**
  - Rationale: do NOT change path while packet burst in progress
  - Proposed in 2008 [Kandula, Katabi, et al, CCR] as a way to optimize load balancing while flows are in progress, without incurring in reordering
- **Input event:**
  - Packet arrival, match TCP port
- **Output actions:**
  - Forward to port/address/tunnel
  - Pick new random port/address/tunnel
- **State:**
  - Currently assigned output (port/address/tunnel)
- **Register update function:**
  - Time stamp + DELTA
- **Condition:**
  - New packet > last packet + DELTA?

Lookup(ip.addr) → state, register T
Condition C: new_timestamp > T
XFSM: if (C=0) forward to port(state);
else forward to port(random/best)
Update: new_timestamp+DELTA → T
port used → state

Giuseppe Bianchi

An OPP program = a platform agnostic abstract XFSM!

(example: a TCP SYN scan detection+mitigation)



Giuseppe Bianchi

Try it yourself

➔**http://openstate-sdn.org/**

⇨Baseline + controller

⇨Based on vanilla Ofsoftswitch → poor performance

➔**http://www.beba-project.eu/open-source**

⇨Latest extensions

⇨Software accelerated Ofsoftswitch → 100x speed

⇨More material

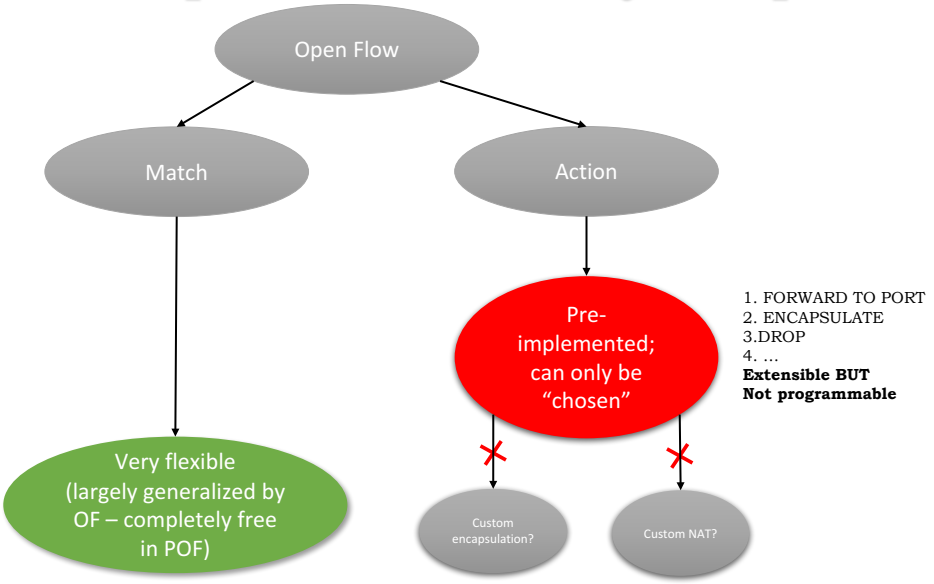
⇨Several use case examples

towards 'true' data plane programmability

Actions → how to make them programmable as well?

- Salvatore Pontarelli, Marco Bonola, Giuseppe Bianchi: "Smashing SDN "built-in" actions: programmable data plane packet manipulation in hardware", IEEE NetSoft 2017, Bologna, Italy, July 3-7, 2017

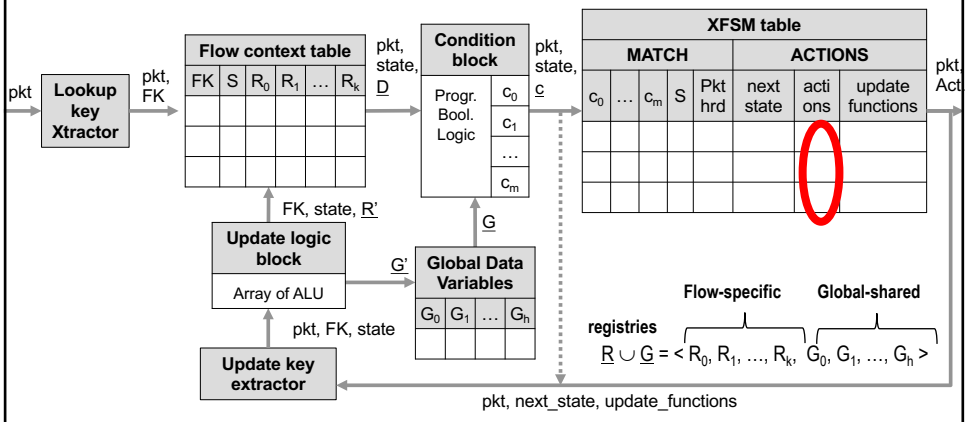
OpenFlow flexibility recap



Giuseppe Bianchi

OpenState, Open Packet Processor

(much) greater flexibility and (stateful) programmability in the match/action association. BUT ACTIONS REMAIN "atomic"



Giuseppe Bianchi

The case for programmable actions

➔**Inband packet reply**

⇨Generate a packet from scratch / from a template?

➔**Tunneling**

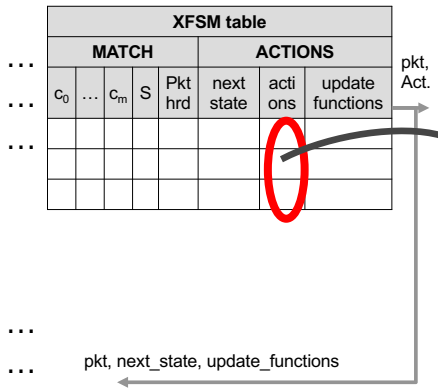
⇨«program» your encapsulation mechanism?

➔Opposed to selecting available tunneling mechanisms

➔**NAT/PNAT**

➔**Etc…**

The case for programmable actions



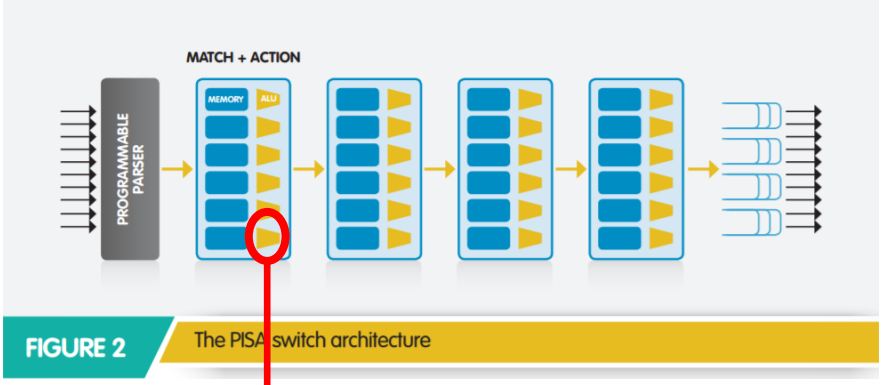
Does not change the overall approach:
Programmable actions at the output of the Open Packet processor

Packet program

Rediscovering the wheel?
Have you heard about P4???
(packet programs is what P4 does!)

Yes, but P4 is «just» a language!
What about the hardware running it?
And is P4 the best approach here?

Digression: P4 over PISA

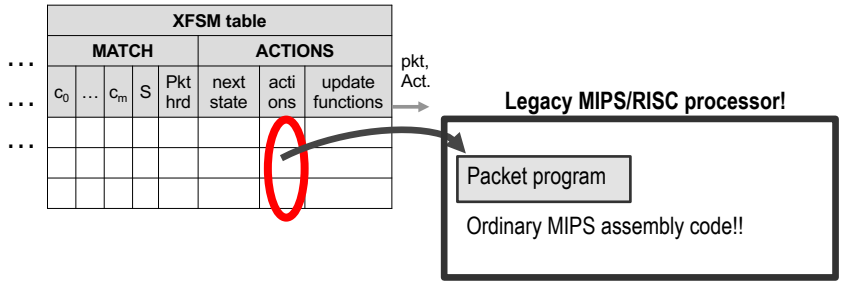


Atomic micro-instruction
(banzai HW – see packet Transactions, Sigcomm 2016)

Multiple μ–instr = multiple stages… (requirement: 1 clock per μ–instr!)
- upper bound on #no μ–instr ? (32 stages)
- is mix of match and μ–instr what we really need?

Occam's razor

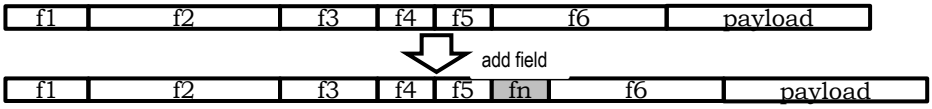
➔Forget about P4 and get back to the «obvious» approach



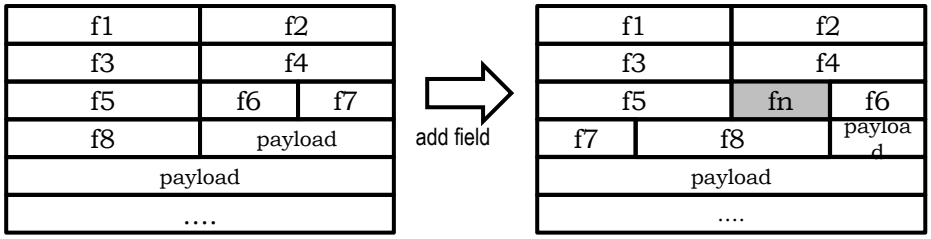
Problem: WAY (!!) too slow ➔ MIPS NOT MEANT to do packet manipulation

Why legacy MIPS are slow?
(mainly because of) Memory management!

Example: if you encapsulate a pkt adding an header field…



… you need to re-align all 32bit memory words – lots of clock cycles!



Our approach

➔ **Take a MIPS IP (VHDL), and strip away all we do not need**

⇨ Small HW footprint

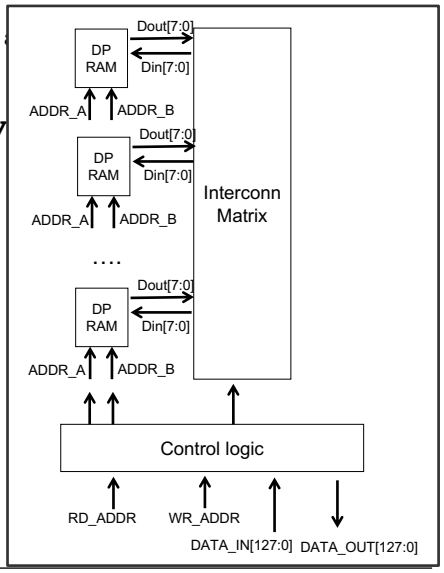
➔ **Improvement #1: change memory management**

⇨ Unaligned memory

Our approach

➔ **Take a MIPS IP (VHDL), and strip**
- ⇨ Small HW footprint

➔ **Improvement #1: change memory**
- ⇨ Unaligned memory



Our approach

➔ **Take a MIPS IP (VHDL), and strip away all we do not need**
- ⇨ Small HW footprint

➔ **Improvement #1: change memory management**
- ⇨ Unaligned memory

➔ **Improvement #2: Add a few new instructions optimizing pkt processing**
- ⇨ Each running in 1 clock
- ⇨ In a normal MIPS they would require multiple micro-instructions

PMP specific	Instruction Type	Instructions	Memory mode	Operands	Note
NO	Logic ALU Operations	NOP, AND, OR, XOR, XNOR,NOT	register-register	rd,rs1,rs2 rd,rs1,imm	standard logic operations
NO	Arithmetic ALU Operations	ADD,ADC, SUB,SBC,MUL,	register-register	rd,rs1,rs2 rd,rs1,imm	standard arithmetic operations
NO	Shift/Rotate Operations	LSL (Logical Shift Left) LSR (Logical Shift Right) ASR (Arithmetic Shift Right) ROR (Rotate Right)	register-register	rd,rs1,rs2 rd,rs1,imm	performs logic and arithmetic shift/rotate operations
NO	Control flow	B(cond) (Branch with condition) BL(Branch and Link with condition) RET (Return), HALT	register-register imm	rs1	The conditions check the status flags and if matched executes the branch. BL also save the pc value in the link register (r14) HALT ends the PMP execution.
NO	Load/Store	ldb, ldh,ldw stb, sth,stw	register-memory memory-register	rd, imm	the operations move 8,16 or 32 bits
YES	mov	movb,movh,movw,movd,movq	memory-memory	rd,addr	movs move up to 128 bits from [addr] to [rd]
YES	out	outb,outh,outw,outd,outq	memory-port	rd, addr	outs move from [addr] to the rd output port.
YES	memory data movement	meml (Memory loop)	memory-memory	rd,rs1,rs2	movl moves rs2 bytes from [rs1] to [rd]
YES	output data movement	outl (output loop)	memory-port	rd,rs1,rs2	moves rs2 bytes from [rs1] to the rd output port

Our approach

➔ **Take a MIPS IP (VHDL), and strip away all we do not need**
- ⇨ Small HW footprint

➔ **Improvement #1: change memory management**
- ⇨ Unaligned memory

➔ **Improvement #2: Add a few new instructions optimizing pkt processing**
- ⇨ Each running in 1 clock
- ⇨ In a normal MIPS they would require multiple micro-instructions

➔ **Results (so far): 10x performance improvement over legacy MIPS**
- ⇨ With lower HW footprint

resource type	standard MIPS	PMP
# Slice LUTs	3634	3169
# Block RAMs	14	14

Our SUME FPGA board: 433.200 LUTs →
1 PMP = 0.7% area

Giuseppe Bianchi

Our approach

➔ **Take a MIPS IP (VHDL), and strip away all we do not need**
- ⇨ Small HW footprint

➔ **Improvement #1: change memory management**
- ⇨ Unaligned memory

➔ **Improvement #2: Add a few new instructions optimizing pkt processing**
- ⇨ Each running in 1 clock
- ⇨ In a normal MIPS they would require multiple micro-instructions

➔ **Results (so far): 10x performance improvement over legacy MIPS**

➔ **Improvement #3 (currently ongoing): improve parallelization**
- ⇨ VLIW: about 2.5x improvement on average
- ⇨ Multiple parallel PMPs per output port: Nx improvements (at the cost of some extra area)
- ⇨ Further custom improvements

Giuseppe Bianchi

Performance over sample Use Cases (before VLIW!)

- **Network Address and Port Translation**
  - 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)

About 30 gbps worst case with VLIW (expected) → still below our ideal 100 gbps target → further custom optimizations

Programmability: assembly (so far)

```
1   .text
2   SendIN:
3    la $1,out_dmac
4
5   #write the ethernet layer
6    la $3,pkt_in       # $3<--in pkt
7    li $2,14
8    outl $op,($1),$2
9
10  #load old IP csum in $12
11   lw $12,24($3)
12
13  #load old UDP csum in $13
14   lw $13,40($3)
15
16  #compute IP delta csum (4 byte)
17   la  $5,source_IP
18   lw  $6,($5)
19   lw  $7,26($3)
20   add $7,$7,$6
21   slr $6,$7,16
22   add $8,$6,$7
23   adc $8,$8,0
24   not $8,$8
25
26  #new IP csum in $8
27   add $8,$8,$12
28
29  #compute L4 delta csum
30   la  $6,source_port
31   lw  $9,($6)
32   lw  $10,34($3)
33   add $11,$11,$10
34   add $11,$7,$10
35   slr $12,$11,16
36   add $12,$12,$11
37   adc $12,$12,0
38   not $8,$11
39
40  #new L4 csum in $9
41   add $9,$8,$13,
42
43  # write fields
44   li $2,10
45   outl $op,14($3),$2
46   outh $op,$8   #write new IP csum
47   outw $op,$5   #write new src IP
48   li $2,4
49   outl $op,30($3),$2
50   outh $op,$6   #write new port
51   li $2,4
52   outl $op,36($3),$2
53   outw $op,$9   #write UDP checksum
54
55  # compute the number of bytes
56   ld   $2,pkt_len   # $3<--pkt_len
57   subi $2,$2,42
58
59  # send data to the output port
60   outl $op,42($3),$2
61
62   halt #halt the PMP
63
64   .data
65  #memory space for the pkt in
66  pkt_in:       .space 2048
67
68  #metadata from parser
69  pkt_len:        .space 4
70  src_mac:        .space 4
71  src_mac4:       .space 2
72  dst_mac:        .space 4
73  dst_mac4:       .space 2
74  eth_type:       .space 2
75  src_ip:         .space 4
76  dst_ip:         .space 4
77  proto:          .space 1
78  src_port:       .space 2
79  dst_port:       .space 2
80  srcip_offset:   .byte 0x1A 0x00 0x00 0x00
81  metadata_space: .space 2017
82
83  op:                     .space 2048
84
85  out_dmac:    .byte 0x00 0x34 0x56 0x78 0x9a 0xbc
86  out_smac:    .byte 0x00 0x43 0x65 0x87 0xa9 0xcc
87  out_type:    .byte 0x08 0x00
88  out_ip:      .byte 0x45 0x00
89  fixed_chk:   .byte 0x12 0xc3
90  source_IP:   .byte 0xA0 0x50 0x50 0x14
91  source_port: .byte 0x10 0x20
92  dest_IP:     .byte 0x90 0xab 0xcd 0xef
93  outer_ttl:   .byte 0x40
94  IP_prot:     .byte 0x04
```

Example: NAT PMP packet program (similar for remaining use cases)

VLIW implementation details

➔ **Just finalized!**

⇨ Brand new IP, started from scratch

➔ **Architecture details**

⇨ 32 bit instructions, 256 parallel load/store data memory per clock

➔(ideal for AXIS NetFPGA-SUME data interface ☺)

⇨ 3 clocks processing (pipelining in progress)

⇨ 8 static parallel lines

➔Up to 8x performance improvement

➔Practical: depends how much you can parallelize your code ➔ manual so far

⇨ Many HW optimizations (heavy prefetching, lane forwarding, etc)

Synthesis (HW) details

Fully synthesized at 250MHz!

Well over 156.25MHz needed for in-out interface!

Clock Summary

Name	Waveform	Period (ns)	Frequency (MHz)
master	{0.000 2.000}	4.000	250.000

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.677 ns	Worst Hold Slack (WHS):	0.030 ns	Worst Pulse Width Slack (WPWS):	1.326 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	27423	Total Number of Endpoints:	27423	Total Number of Endpoints:	7736

All user specified timing constraints are met.

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	3224	0	433200	0.74
LUT as Logic	3224	0	433200	0.74
LUT as Memory	0	0	174200	0.00
Slice Registers	1278	0	866400	0.15
Register as Flip Flop	1278	0	866400	0.15
Register as Latch	0	0	866400	0.00
F7 Muxes	112	0	216600	0.05
F8 Muxes	0	0	108300	0.00

2. Memory

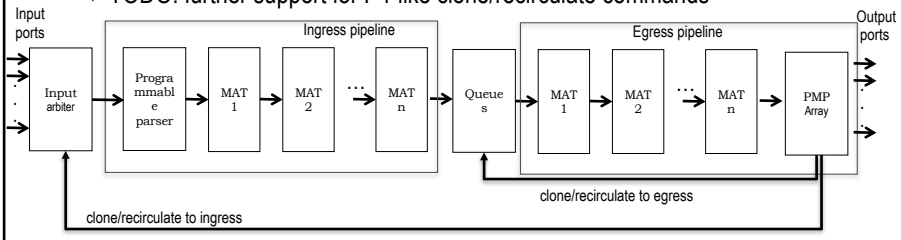
Site Type	Used	Fixed	Available	Util%
Block RAM Tile	44	0	1470	2.99
RAMB36/FIFO*	44	0	1470	2.99
RAMB36E1 only	44			
RAMB18	0	0	2940	0.00

Next steps

➔ **PMP: DONE**
- ⇨ MIPS basic version
- ⇨ VLIW version OK, porting to RISC-V almost done
- ⇨ development of toolchain – in progress
- ⇨ Available (but preliminary, still limited doc/support) @ https://bitbucket.org/marco_spaz/pmp

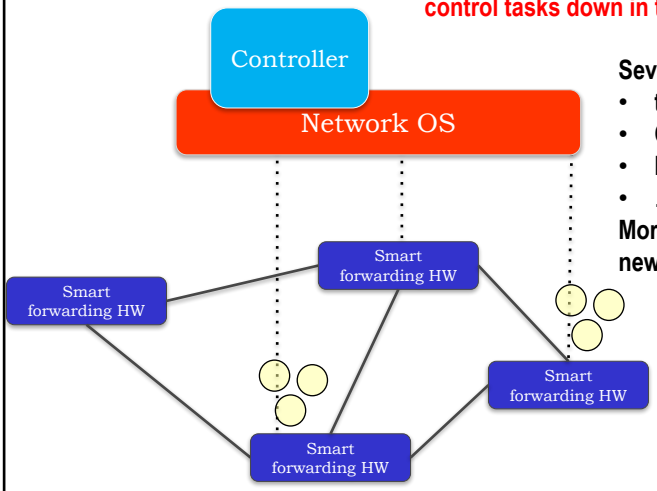
➔ **Add further domain-specific instructions**
- ⇨ E.g. checksum computation in one clock

➔ **Integration in switch architecture**
- ⇨ Just DONE (easy, last stage of pipeline)
- ⇨ TODO: further support for P4-like clone/recirculate commands



Take-home message

Controller still in charge to 'program' the network
But can 'push' time-critical / localized stateful control tasks down in the switches



Several applications
- **traffic policing**
- **Classifiers**
- **DoS mitigation**
- **…**

More ambitious: deploy new protocols!

Conclusions

➔ **Platform-agnostic programming of control intelligence <u>inside devices' fast path</u> seems not only possible but even viable**

- ⇨ «small» OpenFlow extension – OpenState will (most likely) be in 1.6 OpenFlow!!
- ⇨ TCAM as «State Machine processor»
  - →OpenState → Mealy Machines
  - →Open Packet Processor: full XFSM-based processing **without any slow path CPU**
- ⇨ Programmable actions with tailored MIPS/VLIW → custom instruction set for packet manipulation tasks

➔ **Rethinking control-data plane SDN separation?**

- ⇨ Control = Decide! Not decide+enforce!
- ⇨ Openstate/OPP: permit the controller to delegate (local) control programs inside the switches!
- ⇨ Back to active networking 2.0? (but now with a clearcut abstraction in mind)

Giuseppe Bianchi