Received: 7 December 2017 | Revised: 4 April 2018 | Accepted: 12 June 2018

DOI: 10.1002/nem.2043

SPECIAL ISSUE PAPER

WILEY

Smashing OpenFlow's "atomic" actions: Programmable data plane packet manipulation in hardware

Salvatore Pontarelli | Marco Bonola | Giuseppe Bianchi

CNIT/University of Rome Tor Vergata, Rome, Italy

Correspondence
Salvatore Pontarelli, CNIT/University of Rome Tor Vergata, Rome, Italy.
Email: pontarelli@ing.uniroma2.it

Funding information
EU commission; SUPERFLUIDITY project, Grant/Award Number: #671566; 5G-PICTURE project, Grant/Award Number: #762057

Summary

Recently, with new hardware architectures such as Reconfigurable Match Tables and languages such as P4, the Software Defined Networking community has started to bring line-rate *data plane* programmability inside switching chipsets. Starting from the original OpenFlow's match/action abstraction, most of the work has so far focused on key improvements in matching flexibility. Conversely, the "action part," ie, the set of operations (such as encapsulation or header manipulation) performed on packets *after* the forwarding decision, has received way less attention. Goal of this paper is to move beyond the idea of "atomic," preimplemented, actions, and rather make them programmable while retaining high speed multi-Gbps operation. To this purpose, we propose a domain-specific HW architecture, called Packet Manipulation Processor (PMP), able to efficiently implement such actions. Both a PMP C++ instruction set simulator and a NetFPGA prototype have been developed. The performances of the PMP have been verified with three nontrivial use cases (tunneling, NAT, and ARP reply generation), showing that also in the worst case the throughput is well above 10 Gbps.

1 | INTRODUCTION

Software-defined networking (SDN)[1] is arguably one among the most influential innovations emerged in the last years. Even if the ideas behind SDN find roots in early works,[2] it is fair to say that the SDN era was launched by the proposal of OpenFlow[3] as a pragmatic and viable platform agnostic interface to the switch hardware. OpenFlow exposes to the network programmer a so-called match/action abstraction. This abstraction consists in the specification of tables comprising $\{rule, action\}$ pairs; if the rule is matched by the incoming packet, the action associated to the rule is executed. More recently, the need to extend (or even rethink) OpenFlow and make it more flexible and configurable to the programmer's needs has emerged. A hectic research trend has thus started, mainly in the attempt to extend the flexibility of the *match* "part" of the OpenFlow's *match/action* abstraction. Works such as POF (Protocol-Oblivious Forwarding)[4] do significantly improve header matching flexibility and programmability, freeing it from any specific structure of the packet header. Reconfigurable Match Tables, introduced in Bosshart et al[5] permit configuring number, size, and sequence of match tables, providing the hardware support for the emerging P4 higher level language.[6] Finally, works such as OpenState[7,8] (further extended in Bianchi et al[9] to support more general state machines and flow context representations) and FAST[10] explicitly promote matching on per-flow states and perform state transitions and finite state machine execution on the basis of such matches.

Quite surprisingly, with perhaps the exception of some features more recently introduced in the ongoing P4 specification,[11] actions have mainly remained "atomic," and very little work has addressed their flexibility. The program-

 © 2018 John Wiley & Sons, Ltd.

mer can only "select" which action should be associated to the outcome of a match, being such selection restricted to a set of actions (eg, drop, output to port, push/pop VLAN/MPLS tag, etc) preimplemented in the device by the vendor. Conversely, we believe that the ability to flexibly program also the actions is becoming more and more important with the adoption of the SDN paradigm in network middleboxes. In fact, these network elements require to change the content of the packet (eg, for NAT, encapsulation, etc). Furthermore, the ability to program specific packet generation can off-load the SDN controller (eg, the case of Address Resolution Protocol (ARP) reply[12]) or can be used for network measurements and debug.[13]

1.1 | Contribution

The main outcome of this paper is the design of a hardware architecture supporting efficient programmability of actions and designed to sustain high-speed line-rate packet modification/generation. Our proposed *Packet Manipulation Processor* (PMP) is an array of small Reduced Instruction Set Computer (RISC) processors with a specifically devised instruction set able to provide complex packet modification at line rate. In particular,

1 we classify forwarding actions into three main categories: actions that manipulate or insert header fields, actions that forward and dispatch packets, and actions consisting in the forging of new packets triggered by other arriving packets (eg, ARP replies[12]). This classification permits us to identifies the characteristics that the PMP should satisfy to efficiently execute the forwarding actions;
2 we propose a detailed architecture for the PMP, along with the relevant design choices in terms of memory arrangement and interaction with the SDN switch pipeline;
3 two PMP proof of concept implementations: (a) a publicly available PMP software simulator is provided along with the actual implementation and simulation results of three use cases: IP-in-IP tunnel, NAT, and ARP reply packet generation; (b) a preliminary FPGA implementation of a single PMP core inserted in a reference switch pipeline;
4 A performance analysis that shows (a) the maximum sustainable line rate achievable with the PMP and (b) the performance comparison with a standard MIPS CPU (that shows a $x10$ improvement factor in case of a real input packet trace).

A limitation of this paper is the need, so far, to rely on a very low level language to program actions: being, in essence, a microprocessor, the PMP is indeed natively programmed in assembly language, as the applications provided in Section 6 will clearly show. Automatically compile PMP code from a higher level language is thus advocated as future work: while this is in principle very simple, in practice optimization is required to reduce performance impairments (more clock cycles) with respect to direct programming using assembly. Actually, P4 appears to be a natural candidate for such higher level description; indeed some P4 primitives are at least in part already related to our work, although P4 (being a language) does not specify how they are supported in HW whereas this paper specifically focuses on such HW support.

1.2 | Paper structure

The rest of the paper is in structured as follows: in Section 2, an analysis of the possible actions that an SDN switch should perform are presented. In Section 3, we present a reference switch pipeline architecture, while in Section 4, the detailed description of the processor architecture is presented. Section 5 presents the two PMP implementations, and Section 6 show the three use cases along with their performance analysis. In Section 7, a survey of the related work is presented, and in Section 8, the conclusions are provided.

2 | OPENFLOW-TYPE ACTIONS: HW IMPLEMENTATION ISSUES

There is a variety of actions that an SDN device should be able to provide. The availability of these actions in the switch permits to off-load the SDN controller from many repetitive tasks that can be directly applied in the data plane. In this section, we identify the typical actions that a switch must be able to perform, and we will focus our attention on the characteristics of these actions that have an impact on the PMP hardware architecture. This analysis will identify the best design choices of an efficient packet manipulation dedicated processor. We identified three kinds of actions that could be useful to directly implement in the switch, since they can be used to perform many tasks in a wide variety of network protocols.

2.1 | Header field actions

We refer to a first type of actions as *header field actions*. These actions manipulate the content of packet headers by adding, modifying or removing one of the fields composing the header. Examples of these actions are the ones needed to change the DSCP value, to push an MPLS label, to perform IPinIP encapsulation or de-encapsulation, etc. We remark that such operations usually require a single-byte processing granularity. If we suppose that the packet to be processed is stored in a RAM, changing the value of a field (without shifting the remaining part of the packet) requires to read the old value of the memory word containing the field, modify the field value and rewrite the word in the same memory location. Instead, if we need to add a new header field, a more complex operation is required. In Figure 1A, an example of adding one byte as an additional field in the header of a packet is presented. Figure 1B) shows how the packet is stored in a 32 bit memory slot before and after the insertion of the field.

We remark that, even if only one byte is added to the packet, the number of basic memory operations (namely, *read* and *write*) that must be performed is proportional to the packet length, since the whole packet payload must be shifted to create room for the new header. Moreover, since the new header size is most likely not a multiple of the memory word size, the operation to perform on the memory are not aligned. The performance implication of unaligned memory accesses heavily depends on the specific microprocessor architecture. Many microprocessors are able to perform unaligned memory accesses transparently, but there is usually a significant performance cost. Instead, other microprocessor architectures are not capable of unaligned memory accesses, and these accesses must be managed at software level, performing several shift-and-mask operations on the data to read/write. Therefore, it is important to choose a microprocessor architecture that is able to efficiently perform unaligned memory accesses without performance degradation.

Such type of actions can use values coming from different sources. For example, in case of the DSCP field, the value is extracted form the packet itself. Instead, if the action will perform the NAT of a packet, the value to change are stored in one of the MAT of the switch pipeline. Finally, the value can also come from generic switch status registers (such as example to select the output port depending on which ones are failed) or from global variables that the PMP store in the data memory. Therefore, our proposed PMP must be able to read/write data from the following elements:

1. The data memory, that is, the memory that is used to store the packets to manipulate. The same memory can be also used to store some global variables used by the PMP microprograms. This memory is the one with the most bandwidth pressure and require a careful design to maximize the throughput.
2. A set of internal registers. These registers maintain temporary information. Examples of the information stored in these registers are the pointer to the next data to read, the value of a field to be changed in the packet, and an intermediate value of a complex operation.

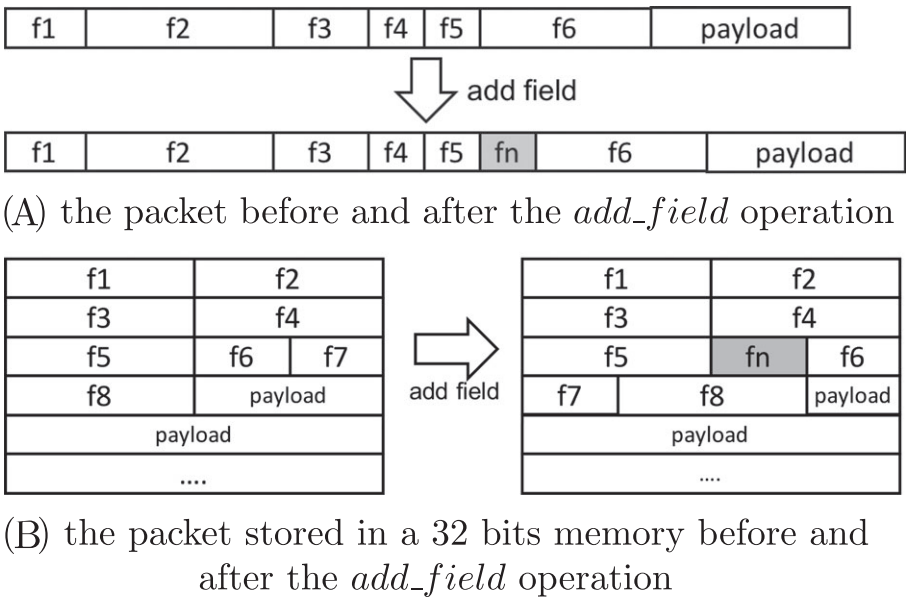

FIGURE 1 The add_field operation

3. The value of the global status registers of the switch. Examples of the information store in these registers are the status of the input/output ports and some global counters/meters, etc.
4. The fields extracted from the packet header. These data can be used as parameter for the action to apply to the packets. Examples of the use of this information are the updating of DSCP value or the parameters to respond to an ARP query. We remark that the same information are also present in the data memory, since they are part of the packet. But, since the extraction operation can be time-consuming, and is already performed to select the keys for the MATs, it is better to reuse the same information, instead of recreate them another time.
5. The outputs of the tables of the processing pipeline. These outputs are the key information of the packet processing pipeline and are used to select the action to apply and the parameters to configure the actions. Examples of the information gathered by this source are the MAC destination address in case of L2 switching, the SRC IP and ports in case of a NAT operation, and the DST IP in case of routing.

2.2 | Dispatching actions

The second type of actions, that we refer to as *dispatching actions* are those related to the moving of the packet in different stages of the processing pipeline and on different output ports of the switch itself. These type of actions require copying or moving the entire content of the packet without modifying the packet content. We can see these operations as a set of read/write from/to the memory in which the packet is stored, to another that can be an I/O memory mapped location (eg, an output port) or another location inside the buffer memory (eg, to implement the P4 clone() operation). From a hardware implementation point of view, such type of operations is easily managed using standard instructions for memory access. However, in order to maximize the performances of these instructions, it is important to underline the following elements:

1. the bandwidth of the memory read/write operation is directly related to the memory data-width and
2. the use of specific instructions that move data from one memory address to another memory address can increase the throughput. This is in contrast with the typical load/store register/memory access of RISC processors, in which the data movement operations require two instructions, one to load the data from the memory to a register, the other to store the data from the register to the new memory location. The implementation of memory to memory operations require dual port (DP) memories, in which one port is used to read data from the memory and the other port is used to write the data in the new memory location. The use of DP memories can also benefit the actions that push/pop a header in the packet.

2.3 | Packet generation actions

Finally, the third type of actions we considered are called *packet generation actions*. While these actions are not common in the current generation switch, they could open many interesting possibility in the evolution of SDN. One application of the in-switch packet generation is to provide low latency response to several types of queries coming from different protocols (eg, ARP reply, heartbeat packets, etc). Another possible application is the use of the switch to perform network analysis, configuring the switch as a packet generator and monitoring the response of the network to the generated packets. The in-switch packet generation relies on the use of predefined packet templates that are stored by the controller in specific memory location. 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 fields to modify are defined by the PMP microprogram and are similar to the field level actions. Indeed, we can see this type of actions as a combination of the first two actions, since the forging of a packet requires to clone the content of a template applying suitable modification to specific parts of the packet. Figure 2 shows the memory corresponding to the initial template and of the actual packet to deliver. F2

In Section 4, the insights described in this section will be used to design the PMP architecture.

3 | SWITCH PIPELINE ARCHITECTURE

In this section, we describe our reference switch pipeline architecture (depicted in Figure 3). While, in principle, it may seem appealing to have a sort of CPU-like "forward processor," we believe that the three steps of the "match/action" abstraction (*parsing*, *matching,* and *action execution*) are extremely different, and therefore using the same HW (a CPU)

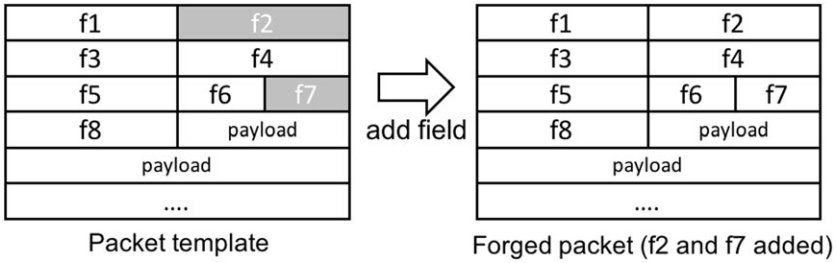

FIGURE 2 Thememory contains the packet template and the forged packet created using the template and the external information gathered by PMP

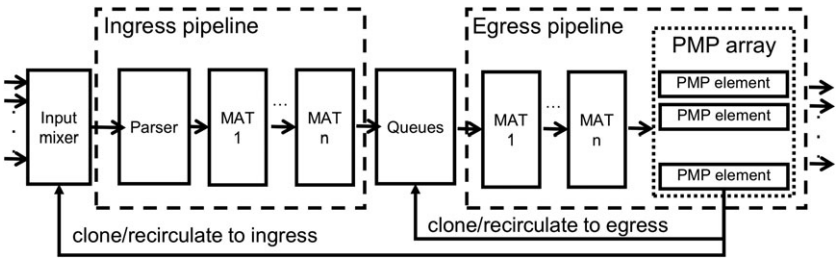

FIGURE 3 Switch pipeline reference architecture. Abbreviations: PMP, Packet Manipulation Processor

to perform these three steps is not an optimal solution. In particular, we believe that the use of specific HW solutions for the first two steps is the best choice. For the first step, we remark that a programmable HW parser can provide both a high level of flexibility and a throughput higher than any processor-based competitor and require only 2% of die area of a switch chip.[14] The header fields extracted by the parser will be available to the PMP battery envisioned at the end of the switch pipeline.

The second step is for sure the most demanding, both in terms of processing requirements and memory footprint. The use of multiple tables further exacerbate this aspect, making mandatory the choice of a hardware-based implementation of multiple MATs. The recent proposals of Reconfigurable Match Tables,[5] as well as the introduction of next generation network chips such as the Intel Flexpipe,[15] proves that an architecture composed by several configurable Ternary Content Adressable Memories (TCAMs) and hash tables can sustain the processing requirements of multiple SDN match operations. Current available TCAMs have significantly higher performance than CPUs executing algorithm based search, except for few specific use cases (such as for example in longest prefix match[16]).

Even if TCAMs have several drawbacks (like high power consumption, limited scalability, and high cost), the research effort is continuously improving the TCAM performance by using the traditional CMOS technology (see, eg,[17] for reducing power consumption or[18] for area efficiency) and novel manufacturing technologies based on resistive or magnetic devices are exploited.[19,20]

Therefore, we propose that only the third step is executed by a CPU-like packet manipulation processor. This design choice results in the following advantages:

1. The instruction set architecture should realize a very specific, homogeneous, and small set of operations. Many of these operations are devoted to memory and I/O data movement. The remaining operations are the typical operation used in a microprocessor, such as the basic logic and arithmetic operation and the standard control flow operation (jumps and conditional branches);
2. The PMP memory hierarchy can be greatly simplified: While the match operation require a CPU with a complex memory hierarchy, the action operations can be efficiently executed by a processor with a flat and easy to realize memory architecture. In fact, the implementation of large hash tables (that can also reach the size of some MB) using a CPU requires a complex memory hierarchy (with several levels of cache, a specific trade-off between the size of each memory level and the memory latency, etc). Instead, the memory required to operate on the packet can be estimated as few KB of code and few KB of data (the processing element should store few packets), thus requiring a flat memory hierarchy. This aspect is somewhat similar to scratchpad memories used in embedded systems,[21] in which instead of implementing complex cache hierarchies, a small and extremely efficient on-chip memory can be used to store the most used data;

3. The proposed architecture permits to define an efficient interface with the rest of the network chip able to gather the information collected by the first two steps of the "match/action" abstraction and manage the movement of the packets in the processing pipeline.

The PMP actions are therefore defined as a sequence of instructions executed by the PMP. We can see this sequence as a microprogram run by the proposed processor. This microprogram will be usually as small as to be stored in a fast RAM and will have several restrictions to guarantee the worst case time. In particular, PMP is not designed to execute complex control flow instructions such as routine calls. When the MAT selects an operation to be executed, it sends to the PMP the memory address in which the corresponding microprogram is stored and launch its execution on the PMP.

Our reference switch architecture, depicted in Figure 3, consists of the following main blocks:

1. An input arbiter that collects the packets coming from the switch input ports:
2. A programmable parser: this block takes as input the packet, identifies its protocol stack a set of packet related data that will be referenced in the subsequent pipeline components (for example, the packet fields composing the flow keys indexing the MAT). The hardware architecture of this block should follow the programmable parser architecture described in Gibb et al.[14] Note that, while the chain of MATs will mainly use the value extracted from the field, the PMP processor requires the knowledge of the offset at which these values are stored, as the field value are already present in the packet and can be easily read by the PMP;
3. An ingress MAT pipeline: the header field and the packet metadata (eg, the switch input port, the time stamp, etc) are sent to a chain of tables to select which actions must be applied to the packet. Depending on the match operation (exact match, longest prefix match, and generic wildcard match), the table can be realized using hash tables, algorithmic LPM search,[16] or a generic TCAM. The output of each table is used both to decide the set of actions, the next table to match and some additional inputs (metadata) for the subsequent MATs. We remark that the subdivision in MATs is defined from a logic point of view, since the same hardware can be used to perform several matches using the same memory;
4. a packet delay queue that store the content of the packet;
5. an egress MATs pipeline;
6. a PMP array: the packet manipulation processor array collects the set of actions to perform and uses the information extracted from the ingress/egress MATs and the packet parser. Each array element is connected to one or more output ports. Virtual output ports provide some feedback loops to implement the packet cloning functions such as (clone and resubmit).

4 | PMP ARCHITECTURE

In this section, the architectural description of PMP and the supported instruction set are described. The overall processor architecture is presented in Figure 4. The architecture is an element of the PMP array depicted in Figure 3.

All packet traversing the switch pipeline are dispatched to the PMP queues, which in turn are connected to one of the CPUs composing the PMP array. Similarly, each single processor is connected to one or more output ports. The packet

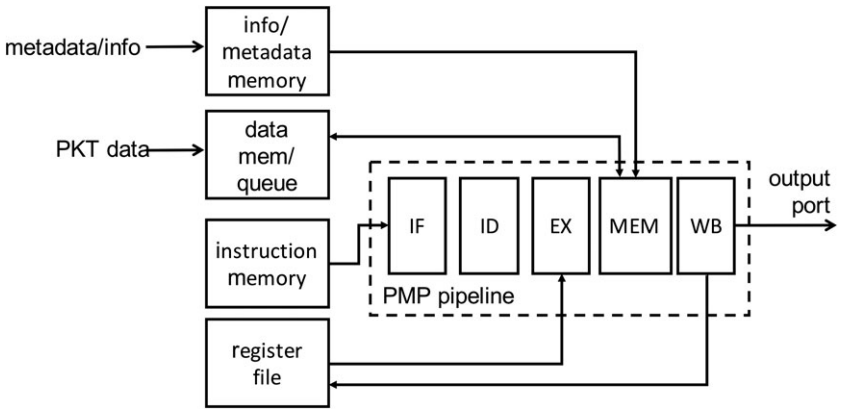

FIGURE 4 Single Packet Manipulation Processor (PMP) element

dispatching to the different PMP array processors is performed within the matching stage, in which the output port for each processed packet is decided.

The architecture of the single CPU composing the PMP array is based on the Harvard architecture in which data and instruction memory are separated.[22]

The single PMP CPU design is based on a simplified five pipeline stage MIPS architecture.[23] In this architecture, the CPU pipeline consists of the following operational stages: the *Instruction Fetch (IF)* stage responsible for reading the instructions from the CPU memory; the *Instruction Decode (ID)* stage that parse the instruction byte code and select the instruction to execute; the *execution unit (EX)* in which the arithmetic logic unit (ALU) executes the selected instruction; the *memory stage (MEM)* responsible for reading/writing data from the data memory; and the *write-back (WB)* stage in which the registers are updated. In the following, we focus on the PMP elements that differ from a standard RISC architecture leaving aside the description of standard RISC elements (instruction memory and register file).

4.1 | Packet information/metadata memory

This 128 bit-wide memory collects the data coming from the switch pipeline, namely, the header fields value, the corresponding packet offsets coming from the programmable parser and the metadata provided by the MATs stages. This information will be used by the PMP to manipulate the packet accordingly to the microprogram in execution.

4.2 | Data memory and queues

The data memory is the critical part of the PMP architecture. The memory should be able to provide both unaligned and aligned data movements, so to minimize the number of clock cycles needed to perform load (memory to register), store (register to memory) and move (memory to memory) operations. In Figure 5, the PMP data memory architecture is reported.

This memory stores the packet processed by the PMP CPUs and consists of 16 eight-bits DP memories. A DP memory is able to concurrently read a data (selected by the ADDR_A address bus) and write another data in another memory location (selected by the ADDR_B address bus). The DP memories are connected using an interconnection matrix that provides one clock cycle data movement. The control logic takes as input the read address RD_ADDR, the write address WR_ADDR, and translate them in the internal ADDR_A/ADDR_B addresses for the DP RAMs. In particular, the four

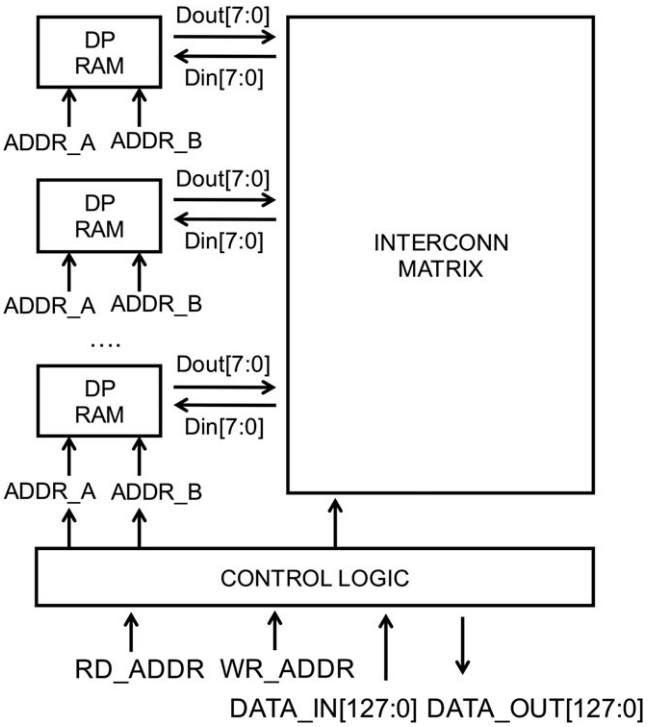

FIGURE 5 Packet Manipulation Processor data memory architecture

least significant address bits of RD_ADDR and WR_ADDR are used to select which DP memory contains the addressed data. Depending on the alignment of the RD_ADDR and WR_ADDR addresses, the control logic and the interconnection matrix selects the memory output data (the Dout signals) and connect them with the corresponding input data.

For example, in case of a 128-bits aligned data movement (eg, RD_ADDR=0 and WR_ADDR=16), each Dout bus is connected to the Din bus of the same memory block, so each byte of the 128 word is copied in another address (ie, is copied from address ADDR_A=0 to ADDR_B=1) of the same memory block. Instead, the copy of an unaligned 128-bits data requires to move the data from a memory block to another. For example, moving the data from RD_ADDR=0 to WR_ADDR=1 require to shift all the Dout buses from the $i - th$ memory block to the $i + 1 - th$ memory block, leaving the same value of ADDR_A and ADDR_B. An exception exists for the $15 - th$ memory block, for which the Dout bus is connected to the Din bus of the $0 - th$ memory block, and the ADDR_B of this block is set to ADDR_A+1.

We remark that the control logic also permits data movement from/to the PMP registers not only limited to 128-bit words but also 8-, 16-, 32-, and 64-bit words by using suitable enable signals (not shown in figure to keep the graphical representation compact). This data movement can be realized using aligned or unaligned addresses. While the RD_ADDR is privately held by the PMP, the WR_ADDR is shared between the PMP and the control logic block that writes the packet data coming from the switch pipeline. In particular, when the PMP does not write on the data memory, the control block fetches the data from the switch pipeline. The memory space allocated for the queue can be dimensioned in order to exclude a portion of the memory packet storing. This memory space can be used to maintain persistent information or to store predefined packet templates to use for packet generation.

Finally, we designed the PMP architecture to manage the output ports connected to the PMP as specific memory locations in which the data can be written (using the 128-bits data bus).

4.3 | Arithmetic logic unit

The PMP has a basic Arithmetic Logic Unit (ALU) that executes arithmetic and bitwise logical operations. Depending on the selected OPCODE, the ALU performs bitwise AND, OR, XOR, NOT logical operations, arithmetic operations such as Addition, Subtraction, etc, and logical shift/rotate operations. The two input operands of the ALU are taken from the register file or is an immediate value taken from the instruction to execute, and the output of the selected ALU operation is stored in the register file. We remark that, differently from the MOV instructions, ALU operations cannot directly operate on data memories but only register to register operations are allowed. This choice is due to performance reasons, since a path going from/to the memory thru the ALU should negatively affect the maximum operating frequency of the PMP.

The ALU also contains a standard set of status flag registers (Carry, Zero, Negative, Overflow) that are set by the result of the ALU operation. These flags can be used in the subsequent ALU operations or for controlling conditional branching.

4.4 | Instruction set

The complete PMP instruction set is reported in Table 1. The subset of the ALU operations (logic, arithmetic, shift, and compare) is a typical set of RISC instructions. These instructions use two operands taken from the register file or one operand taken from the register file and an immediate operand encoded in the instruction.

The control flow operations is minimal and performs the basic control flow tasks. The main difference of the PMP from a minimal RISC processor is the support for movement instructions. The PMP can operate using both register to memory operations (the typical load and store instructions of the RISC architecture) and memory to memory operations. Data movements from/to memory can be performed over different data width spaces. In particular, load/store instructions can move 1, 2, and 4 bytes and *mov* instruction can move 1, 2, 4, 8, or 16 bytes. A specific set of "output" operations are used to move the data in the PMP queue to the physical or virtual output ports. These ports are identified by the destination register of the *out* instruction. Instructions *movl* (move loop) and *outl* (out loop) are able to perform data bulk movement with a throughput of 16 bytes for clock cycle. It is worth to notice that a standard RISC architecture could require three or four clock cycles for the loop execution.

4.5 | P4 action translation

In this section, we present a feasibility analysis of the PMP ability of executing the P4 action primitives. It is worth to notice that we do not developed a compiler from P4 actions to PMP instructions (this compiler should probably

TABLE 1 PMP instruction set

Instruction type	Instructions	Memory mode	Operands	Note
Logic ALU	NOP, NAND, AND, OR,	register-register	rd,rs1,rs2	standard logic operations
Operations	NOR, XOR, XNOR, NOT		rd,rs1,imm	
Arithmetic ALU	ADD,ADC,	register-register	rd,rs1,rs2	standard arithmetic operations
Operations	SUB,SBC,MUL,		rd,rs1,imm	
Shift/Rotate	LSL (Logical Shift Left)			
Operations	LSR (Logical Shift Right)	register-register	rd,rs1,rs2	performs logic and arithmetic
	ASR (Arithmetic Shift Right)		rd,rs1,imm	shift/rotate operations
	ROR (Rotate Right)			
	B(cond) (Branch with condition)			The conditions check the status flags and if
Control flow	BL(Branch and Link with condition)	register-register	rs1	matched executes the branch.
	RET (Return),	imm		BL also save the pc value in the link register (r14)
	HALT			HALT ends the PMP execution.
	ldb, ldh,ldw	register-memory	rd, imm	the operations move 8,16 or 32 bits
Load/Store	stb, sth,stw	memory-register		
mov	movb,movh,movw,movd,movq	memory-memory	rd,addr	movs move up to 128 bits from [addr] to [rd]
out	outb,outh,outw,outd,outq	memory-port	rd, addr	outs move from [addr] to the rd output port.
memory data movement	meml (Memory loop)	memory-memory	rd,rs1,rs2	movl moves rs2 bytes from [rs1] to [rd]
output data movement	outl (output loop)	memory-port	rd,rs1,rs2	moves rs2 bytes from [rs1] to the rd output port

Abbreviation: ALU, Arithmetic Logic Unit; PMP, Packet Manipulation Processor.

TABLE 2 P4 primitive actions and their PMP mapping

Primitive	Notes
	Field actions
add_header	the packet stored in the PMP memory queue is shifted down by *header.length* bytes to provide room for the new header. All the bytes of the created space is set to zero.
copy_header	a bulk move in the PMP memory queue is performed.
remove_header	the packet stored in the PMP memory queue is shifted up by *header.length* bytes.
modify_field	write a value in the PMP memory location identified by a *dest* parameter
modify_field_with_hash_based_offset	the field is modified according to a specific function, defined by a field list calculations parameter. Some function can be easily computed by the PMP (eg, xor16,csum16,optional_csum16). Others require specific ISA extension (see, eg, crc16,crc32)
push	add an array of headers
pop	remove an array of headers
count	increment a counter and store the persistent area of the PMP memory
meter	execute a meter operation and store the persistent area of the PMP memory
	Dispatching actions
drop	do nothing.
truncate	copy *N* bytes of the packet stored in the PMP memory queue to the output port
resubmit	the packet stored in the PMP memory queue is moved to the ingress virtual port
recirculate	the packet stored in the PMP memory queue is moved to the egress virtual port
clone_ingress_pkt_to_ingress	the packet stored in the PMP memory queue is copied to the ingress virtual port
clone_egress_pkt_to_ingress	the packet stored in the PMP memory queue is processed and copied to the ingress virtual port
clone_ingress_pkt_to_egress	the packet stored in the PMP memory queue is copied to the egress virtual port
clone_egress_pkt_to_egress	the packet stored in the PMP memory queue is processed and copied to the egress virtual port
generate_digest	This primitive action is defined in P4 to provide a generic mechanism to send data to an external receiver that performs specific processing a portion of the packet. This receiver can be anything from a fixed function piece of hardware to a control-plane function. The PMP can support this primitive in two ways: 1) directly applying the algorithm to the specified packet portion or 2) sending out the packet using the specific virtual port for the control plane.

leverage on a C compiler as an intermediate step), but we only limit to show that the PMP is able to execute all the P4 action primitives defined in the language. For sake of concreteness, in Table 2, we reported the set of P4 primitive actions taken from the P4 Language Specification Version 1.1.0[11] and discussed how they can be converted into PMP instructions.

The table is divided in two sections for the two action types identified in Section 4. The first type of actions refer to the header and field manipulation actions. The second type of actions dispatch the packet to virtual or physical ports. We considered generate_digest and truncate as dispatching actions since they require to process the packet payload or the entire packet. Note that P4 does not support in switch packet generation primitives. To better understand how P4 actions are mapped into PMP routines, we selected a subset of the action and reported in Section 6 the corresponding assembly code.

5 | PMP IMPLEMENTATION

In this section, first, we describe a C++ based simulator implemented to identify and asses the best minimal instruction set and to gain insight on the performance improvement of the PMP architecture. Second, we describe a preliminary FPGA prototype of a PMP core and its integration within the reference switch pipeline. The aim of the FPGA prototype is to understand the hardware feasibility of the PMP architecture and to evaluate the logic resources cost overhead with respect to the reference switch pipeline. Finally, we discuss the cost and performance achievable using an ASIC technology. SW simulator. CPU Instruction Set Simulators are widely used in the computer architecture research community (see, eg, Burger et al[24]) to simulate the performance achievable with a specific microprocessor architecture. The PMP simulator

TABLE 3 Hardware cost of standard MIPS and the PMP

Resource type	Standard MIPS	PMP
# Slice LUTs	3634	3169
# Block RAMs	14	14

Abbreviation: PMP, Packet Manipulation Processor.

TABLE 4 Hardware cost of the reference pipeline with and without one PMP for each output port

Resource type	OpenState switch without PMP	OpenState switch with four PMPs
# Slice LUTs	71 712 (16%)	84 392 (19%)
# Block RAMs	393 (26%)	437 (29%)

Abbreviation: PMP, Packet Manipulation Processor.

has been implemented as an extension of a simple and compact publicly available MIPS CPU simulator[25] and provide full support for the PMP instruction set described in this paper. The PMP simulator consists of three components:

1. a microprogram assembler that takes an assembly PMP microprogram and generates the PMP binary executable. This component has been extended in terms of its syntax and semantic in order to parse the new PMP assembly instructions;
2. a microprogram Instruction Set Simulator that takes as input the PMP microprogram byte code and performs the packet actions described in the PMP binary executable. This module has been extended to support the execution of the newly introduced PMP instructions;
3. a PMP packet interface module responsible for (a) taking PCAP traces as input and writes a set of packet metadata in the data segment of the PMP microprogram (field values and offsets for Ethernet, IP, and transport layer headers) and (b) writing in an output PCAP trace the packet processed by the PMP and dispatched to the output queue.

The PMP simulator is able to reproduce the CPU cycles accurately and permits to measure the number of clock cycles needed to execute a PMP microprogram. From this, we were able to understand useful information without the need of a detailed hardware design, like the estimation the throughput achievable by the 1 GHz, five-pipeline stage PMP.

The PMP source code, the assembly code of the use cases described in the following subsections, and a simple guide to generate the PMP executable and execute the simulations are available at the PMP project public repository.[26] The PMP hardware prototype has been implemented on the NetFPGA SUME,[27] a x8 Gen3 PCIe adapter card incorporating a Xilinx Virtex-7 690T FPGA, four SFP+ transceivers providing four 10GbE links, three 72 Mbits QDR II SRAM, and two 4GB DDR3 memories. The FPGA is clocked at 156.25 MHz, with a 64 bits data path from the Ethernet ports, corresponding to 10 Gbps per port.

First, we synthesized a standard MIPS and a single PMP with the specific instructions described in Section 4. The comparison of the two implementations in terms of number of resources (Slice LUTs) is reported in Table 3).

As expected, the PMP implementation requires less logic resources since we removed the multiplier and divided that are available in the standard MIPS architecture. Then, we have integrated a PMP array at the end of a pipeline of two OpenState stages[8] that correspond to a pipeline of two MAT stages of the reference architecture described in Section 3. For each output port, a PMP core has been instantiated.

Table 4 reports the logic and memory resources (in terms of absolute numbers and fraction of available FPGA resources) used by the reference switch FPGA implementation without the PMPs and compare these results with those obtained when the PMPs are added. As expected, the PMP array uses a small fraction of the total area (the increase with respect the switch without the four PMP cores is 3% of the FPGA resources).

ASIC viability. To estimate the feasibility of our proposed architecture when implemented using a last generation ASIC technology (eg, a process node of 22 nm), we consider a PMP array deployed in a 64 ports × 10 Gb/s programmable switch. This is similar to the Intel FlexPipe FM6000 ASIC chip[15] as well as to the switch described in Bosshart et al.[5] Supposing a PMP operating frequency of 1 GHz we can suppose a simple configuration with one

PMP element for each output port. Therefore, we need 64 PMP elements for the whole switch chip, and each PMP must provide a minimum throughput of 10 Gbps to operate at line rate without packet loss. We can estimate the area cost of a PMP element around 20 K equivalent NAND gates.* Also the memory blocks are extremely small, since the register file only requires 512 bits of memory, and the data and instruction memories are in the order of few KBs. Even if with a pessimistic assumption of 20 K equivalent NAND gates also for these memories, the overall area of the PMP array is around 2.5 M gates. Considering that the overall area of a switch chip is in the order of 10^8, the area of the PMP array is less than 2.5%. This estimation is inline with the overhead estimated for the FPGA implementation.

6 | USE CASES AND PERFORMANCE ANALYSIS

In this section, we first introduce some simple examples of PMP routines able to execute P4 action primitives and after we present the three complete application use cases. The P4 actions are used to introduce the features of the PMP while the use cases were used to test the functionalities and the performance of the PMP and its comparison with the performance achievable with a standard MIPS architecture.

6.1 | Implementation of P4 actions

To introduce the reader to the way in which PMP routines are implemented, we start from some simple examples that implements the P4 actions presented in Table 2. The first code example is presented in Figure 6.

Two P4 actions are taken into account, namely, the push and the modify_field primitives.

The assembly program consists of two parts. The first part is the actual code to be executed (the text section), while the second part is the data section where the memory locations of the data used by the program are defined. The code segment have some labels that correspond to the function entry point. In the specific case, the two entry points correspond to the P4 actions that the PMP can execute. Depending on the match output of the pipeline, one of the entry point is selected, and the corresponding action is executed.

Instead, the data section defines the memory location in which the program will retrieve the internal memory, the packet memory (where the packet payload is stored), and the information provided by the switch pipeline to the PMP. In particular, the data section defines 2KB for the incoming packet, 2KB for the metadata information, and 2KB for the internal program data. In the specific case, the metadata are the count parameter of the push primitive and the dest, value, mask parameters of the modify_field primitive.

The push primitive takes an array of bytes, representing the headers of the packet and moves the data contained into the array down of *count* bytes, making space for new elements. This primitive is realized moving the memory data stored in the data segment using the memory loop instruction. Instead, the modify_field primitive updates the field defined by the *dest* parameter using a masked value. The *dest* and *value* parameters are taken from the pipeline while the *mask* parameter in this example is specified as a constant and therefore is stored in the internal data section.

6.2 | Use cases description

In the remainder of this section, the following three application use cases are described: (1) Network Address and Port Translation (NAPT); (2) ARP reply; (3) IPinIP packet encapsulation.

It is worth noting that these examples could have been expressed in terms of higher level language action primitives. A direct mapping between these primitives and the corresponding sequence of assembly instructions could have been sufficient to provide a working program. However, such direct mapping provides a nonoptimized assembly code. For example, consecutive action primitives could access to the same memory location, or to contiguous memory locations. These accesses could be performed as a single memory access. The optimization of the code directly derived from the action primitives requires an optimization algorithm almost identical to the back-end optimization phase of a C/C++ compiler. Since we target this optimization as a future work, now, we preferred to develop the use cases directly in assembler. This allowed us to better estimate the minimum number of clock cycles required by each application (and the relative throughput against the input packet length).

*ASIC engineers usually use the number of NAND2 equivalent gates to abstract from the manufacturing process node.

```
# .text segment (program code)
 .text


## entry point: start of function
## push(array, count)
push:

# fetch the count value into the PMP register r1
  lw $1,count

# fetch the array pointer into the PMP register r2
  la $2,array

# compute the destination pointer (array+cont) in r3
  add $3,$2,$1

# set the number of bytes to move in r4
  li $4,128

# move the array down of count bytes using the
# Memory loop instruction
  movl $3,$2,$4

# exit: halt the PMP
exit_push:
  halt

## entry point: start of function
## modify_field(dest, value, mask)
modify_field:

# fetch the metadata into the PMP registers
  lw $1,dest
  lw $2,value
  lw $3,mask

# apply the mask
# dest= (not (current_value & mask)) | (value & mask)
  nand $4, $1,$3
  and  $5, $2,$3
  or   $6, $4,$5

# write the computed value into the dest field
  stw $6,dest

# exit: halt the PMP
exit_modify_field:
  halt

## data segment
.data
#memory space for the pkt in
pkt_in:      .space 2048

#metadata from pipeline
count:              .word  16
dest:               .space 4
value:              .space 4

metadata_space: .space 2036

# internal data
array:          .space 128
mask:           .word 0xff0000ff
internal_space: .space 1916
```

FIGURE 6 Packet Manipulation Processor microprograms for P4 actions

```
# .text segment (program code)
 .text
SendIN:
  la $1,out_dmac

#write ethernet header (14 bytes)
  la $3, pkt_in
  li $2,14
  outl $op,($1),$2  #out loop

# compute the IP csum
  lw $6,14($3)
  lw  $7,18($3)
  add $8,$7,$6
  lw  $6,22($3)
  add $8,$8,$6
  lw  $6,26($3)
  add $8,$8,$6
  lw  $6,30($3)
  add $8,$8,$6
  slr $7,$8,16  #get the 16 msb
  add $8,$8,$7
  adc $8,$8,0
  not $8,$8     #final csum is in $8

  li $2,10
  outl $op,14($3),$2
  outh $op,$8  #write csum
  outw $op,src_ip
  li $2,4
  outl $op,30($3),$2
  outh $op,src_port

# compute the number of bytes
# from IP seq num to eop
  ld   $2,pkt_len   # $3<--pkt_len
  subi $2,$2,36 # subtract addr and port

# out the packet payload to the out port
  outl $op,36($3),$2

# exit: halt the PMP
exit:
  halt

## data segment
.data
#memory space for the pkt in
pkt_in:      .space 2048

#metadata from parser
pkt_len:        .space 4
src_ip:         .space 4
src_port:       .space 2

# program arguments
metadata_space: .space 2038

# internal data
out_dmac: .byte 0x00 0x34 0x56 0x78 0x9a 0xbc
out_smac: .byte 0x00 0x43 0x65 0x87 0xa9 0xcc
out_type: .byte 0x08 0x00
internal_space: .space 2024
```

FIGURE 7 Network Address and Port Translation Packet Manipulation Processor microprogram

6.2.1 | Network address and port translation

A NAPT application requires the following packet header field translation: (1) the source IP and transport port for packets coming from the internal network, as in the case, for example, of dynamic NAT from a masqueraded LAN; (2) the destination IP address and transport port of packets coming from external networks, as in case the case, for example, of static port forwarding. The actual choice of the specific NAT transformation is driven by the first two switch pipeline stages and in particular by the rules inserted by an external SDN controller.

In either cases, the switch retrieves from a NAT binding table (that is implemented in one of the available MAT stages), the value of the (IP, port) pair and the makes it available to the PMP processor. The task of allocating a new entry in the NAT binding table for each new connection is again delegated to an external SDN controller. In particular, when the first TCP/UDP packet of a flow coming from the private network arrives, it triggers the activation of the SDN controller. For example, in case of dynamic NAT, the controller decides the association between the private (IP, port) pair and the public (IP, port) pair and inserts this new entry in the NAT binding table.

From this point, the pipeline stages provide to the PMP which substitution must be performed (ie, which function must be executed) and which data must be inserted in the packets. The PMP executes one of the functions presented in Figure 7 and send the packet to the output port attached to the PMP. The SendIN() function is used to translate the packet from the external to the internal network. Due to the limited space available in this paper, the NAPT microprogram does not show the SendOUT() function (makes the translation in the opposite direction) and only refer to UDP packet translation (TCP NAT operations would require a different code for the transport layer checksum re-computation).

The code rewrites the ethernet header, recompute the checksum, overwrites the SRC IP address (src_ip) and UDP source port (src_port), and finally copies the packet payload to the output port. The data section contains the metadata coming from the pipeline (pkt_len, src_ip, src_port).

The throughput achievable for this application depends on the average packet length. The worst case corresponds to minimum size packet, that is the 64 byte minimum Ethernet frame, corresponding to four words of 128-bits. For this minimum size packet, the code is executed in 44 clock cycles, providing a minimum throughput of around 11.6 Gb/s. The maximum throughput achievable occurs when all the packets has the maximum length. Supposing an MTU of 1500 bytes, the code is executed in 133 clock cycles per packet, corresponding to a throughput of 90,2 Gb/s.

```
1   .text
2   ARPreply:
3   ## write the ethernet header
4   ##variables form data segment
5       outwh $op, src_mac
6       la $1, sender_mac
7       li $2,22
8       outl $op, ($1),$2
9
10  #write the ARP response
11  #variables from data segment
12      outw $op, dst_ip
13      outwh $op, src_mac
14      outw $op, src_ip
15
16  #padding (64 bytes)
17      la $1,padding
18      li $2,18
19      outl $op, ($1),$2
20
21  # exit: halt the PMP
22  exit:
23      halt
```

FIGURE 8 Address Resolution Protocol (ARP) reply Packet Manipulation Processor microprogram

```
1  .text
2  encapsulate:
3  ## compute total\_len
4     sth  $2,2(pkt_start)
5     add  $1,$2,20
6
7  #check for fragmentation:
8     subi $r1,$r1,MTU    # if (tot_len>MTU)
9     bge  fragment       # jmp to fragment
10
11 #write unfragmented packet
12 [..]
13 #  write the ethernet layer
14 [..]
15 #  write the outer IP layer
16 [..]
17
18 # Header Checksum computation. It uses
19 # Precomputed csum of the fixed part and
20    movw $5,fixed_chk    # initialize the csum
21 [..]
   # update the csum
22    outh $op,$6      # write the outer IP csum
23
24 # decrease the TTL of the inner IP packet
25    lb  $5, 8(pkt_start) # read the TTL
26    subi  $5,$5,1          # decrease the TTL
27    bz    drop             # drop if TTL = 0
28    outb  $op,$5           # write the new TTL
29 [...]
30
31 exit:
32     halt
33
34 # fragment: execute fragmentation
35 fragment:
36 [...]
37 # drop: drop if TTL =0
38 drop:
39 [...]
```

FIGURE 9 IPinIP PMP microprogram

6.2.2 | ARP reply generation

The ARP reply use case is useful to show how the PMP can generate a new packet starting from the information gathered from an incoming packet (the ARP request packet) and from the MATs stages of the switch pipeline. The snippet of the ARP reply code is shown in Figure 8.[†]

The code is executed each time the switch pipeline identifies an ARP request. The parser of the switch pipeline extracts the requested IP and a MAT stage provides the MAC associated to the requested IP. The target MAC and IP and the sender IP are provided to the PMP as the variable parts of the ARP reply packet. Instead, The constant information of the ARP reply packet, (namely, the source MAC of the responder, Ethernet type, HW and protocol type and size, ARP opcode, and sender MAC) are stored in the .data section from the address labeled sender_mac. The ARPreply code is always executed in 18 clock cycles, which correspond to a throughput for this application of around 28.4 Gb/s considering the minimum size packet of 64 bytes.

[†]The data section is not shown.

TABLE 5 Comparison of the use cases on standard MIPS and PMP: clock cycles and throughput in Gbps (between parenthesis)

Use case	PMP min size	PMP avg	MIPS min size	MIPS avg size
1	44 (11.63)	100 (30)	110 (4.65)	911 (3.29)
2	18 (28.44)	-	53 (9.66)	-
3	42 (12.19)	98 (30.6)	106 (4.83)	902 (3.32)

Abbreviation: PMP, Packet Manipulation Processor.

6.2.3 | IPinIP encapsulation

IP in IP is an IP tunneling mechanism that encapsulates one IP packet in another IP packet payload. The application described in this section inserts an outer header in the packet, adding a new source and destination IP addresses (the tunnel endpoints). The inner packet is unmodified (except the TTL field, which is decremented). If the packet size is greater than the link MTU, the packet is fragmented, following the procedure described in RFC 791. The code snippet of the IPinIP application is shown in Figure 9.

We selected this encapsulation to show the versatility of the PMP processor. The tunneling requires several operations: updating the TTL fields, recomputing the checksum of the inner and outer IP packets, managing the fragmentation, etc. The proposed code covers the real tunneling mechanism, considering all the options that can occur in the IPinIP encapsulation approach.

The code first checks if the packet must be fragmented. If the packet size is less than the MTU, the routine encapsulates the incoming packet in an outer IP packet. The PMP builds the header fields of the outer IP pack et also using the information extracted from the inner packets. In particular, the routine gathers from the inner packet the total packet length, the TOS and the "don't fragment flag." After, the routine recomputes the IP checksum and writes the composed header to the output port. Finally, it decrements the TTL of the inner packet, updates the inner IP header checksum, and copies the packet content to the output port. Instead, if the packet size exceeds the MTU, the PMP jumps to the fragment procedure. This procedure, not shown for sake of brevity, is a loop that execute a sequence of instructions similar to the one described for the unfragmented IP packets.

Also in this application, the worst case occurs with the minimum size packet (four words of 128 bits). For this minimum size packet, the code is executed in 42 clock cycles, providing a minimum throughput of 12.2 Gb/s.

We remark that the packet exceeding the MTU, that must be fragmented by the PMP requires a more complex execution, but the actual throughput is much less demanding, since the execution cost is mitigated by the great amount of data transferred by PMP.

6.3 | Performance comparison with standard MIPS

To confirm the advantages of our proposed architecture we implemented the micro programs described in the previous sections also using only the standard MIPS instruction set. In Table 5, we report the number of clock cycles and the throughput achievable with a standard MIPS and with PMP. The evaluations for the NATP and IPinIP use cases consider a real packet trace of 30 seconds captured at our campus network (the traces has 83k IP packets with average length of 375 bytes and 6523 unique socket five-tuple) and a synthetic trace of minimum size packets (64 bytes). The ARP use case is independent from the input packet size as the generated ARP replies have a fixed length of 28 bytes and therefore the throughput for average packet size is not reported. The table clearly shows the advantage of the PMP instruction set with respect to the standard MIPS. The improvement is around 3x for the minimum packet size and close to 10x for a real trace.

7 | RELATED WORK

In this section, we present an overview of the related work, organized in three categories. First, we present some recent works that propose domain-specific language for SDN and discuss the interaction with the proposed PMP architecture. Then, we present work that could exploit the flexibility of the PMP to add new features to the switch. Finally, we give an overview of other programmable hardware architectures and compare them with our proposed architecture.

Domain Specific Languages. The mostly influential papers in the field of DSL for programming packet processing are P4[6] and POF (Protocol-Oblivious Forwarding).[4] Both these languages allow defining the protocol parsing graph and heavily exploit the MAT stages to select the actions to apply to the packets. As previously discussed, the PMP has been designed exactly to implement the set of P4 primitives and the POF Forwarding Instruction Set. All the primitives defined in P4 and the Forwarding Instruction Set defined in POF are easily implementable using the PMP architecture. Thus, our proposal is extremely useful to implement a line rate programmable switch that fully support the action set required by the above mentioned DSLs. In fact, it is possible to directly write the P4/POF instructions as PMP functions.

Application frameworks. OpenFlow[28] specifies some simple, very specific actions, such as push/pop of specific labels (VLAN, MPLS, and PBB), copy or decrement TTL, or set a specific value to a packet field. Implementing these actions on PMP is straightforward.

A method to generate packets is proposed in Bifulco et al,[12] which proposes a framework to specify how a switch can generate packets directly in the SDN switch, avoiding communication exchange with an SDN controller. A compiler could easily provide the translation from the InSPired representation to an executable for PMP.

Jeyakumar et al[29] propose to insert directly in the packet very small programs that can be executed by the switch during the travel of a packet in a network. The actions proposed in Jeyakumar et al[29] can be easily realized by PMP. In particular, the idea of data exchange between network devices injecting data into the packets in a space called "Packet Memory" can be directly managed by PMP using the data memory/queues.

Competitor architectures. One option for design programmable switches is based on software architectures running on general purpose CPU or on GPU. Examples of this approach are RouteBricks,[30] PacketShader,[31] and CuckooSwitch.[32] This choice has a great flexibility but also imply several limitations. The most important ones are the limited throughput, the switch latency, and the jitter.

Another option is the use of specific network processor units such as the Cisco QuantumFlow[33] or the EZchip NP-5.[34] These architectures have higher throughput than the one based on generic CPU/GPU but are more complex to program. Moreover, they rely on a complex memory hierarchy and interconnection, that is absent in our proposed PMP architecture. Finally, even if network processor units can be equipped with internal/external TCAM, the processor dominates all the steps of the packet processing. Instead, the PMP follows a pure ASIC switch pipeline for protocol parsing and decision-making steps and rely on the software execution only for packet modification.

Also FPGAs have serious limitations. In particular, the hard programming model based on hardware description languages,[‡] the limited memory amount, the absence of efficient wildcard matching structures like TCAM, and the excessive cost of these devices.

Finally, several data plane forwarding architectures such as Reconfigurable Match Tables[5] or Flexpipe[15] apply the action to the packet using a chain of use a chain of elementary blocks. Our PMP can be used to substitute these chains, or in parallel to the action chain to enhance action flexibility of these programmable switches.

8 | CONCLUSIONS AND FUTURE WORK

In this paper, we discuss how to extend the SDN programmability providing software defined actions. First, we identify the type of actions that are useful for network programmability, after we selected a specific instruction set of a RISC microprocessor to maximize its performance with respect to the expected set of packet manipulation functions. The details of a hardware architecture that exploit the specific instruction set has been shown. Finally, a set of use cases has been proposed to show the effectiveness of the proposed architecture. Possible future works include: (1) the design, implementation, and synthesis of a HW FPGA prototype of the PMP CPU array; (2) the design and implementation of a higher level language compiler; and (3) the integration of the PMP SW implementation within existing SDN SW switches (openflow, P4).

ACKNOWLEDGEMENT

This work has been partly funded by the EU commission in the context of the SUPERFLUIDITY project, Grant Agreement #671566 and 5G-PICTURE project, Grant Agreement #762057.

[‡]We are aware of tentative to develop high level languages for FPGA tailored to SDN, but they are at early stages and often greatly worsen the performance achievable with respect to a pure hardware description languages design.

ORCID

Salvatore Pontarelli http://orcid.org/0000-0002-3626-6404

REFERENCES

1. Greene K. TR10: Software-defined networking. *MIT Technol Rev*. 2009. http://www2.technologyreview.com/article/412194/tr10-software-defined-networking.
2. Feamster N, Rexford J, Zegura E. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Comput Commun Rev*. 2014;44(2):87-98.
3. McKeown N, Anderson T, Balakrishnan H, et al. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Comput Commun Rev*. 2008;38(2):69-74.
4. Song H. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN '13; 2013:127-132.
5. Bosshart P, Gibb G, Kim H-S, et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In: ACM SIGCOMM 2013; 2013:99-110.
6. Bosshart P, Daly D, Gibb G, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Comput Commun Rev*. 2014;44(3):87-95.
7. Bianchi G, Bonola M, Capone A, Cascone C. Openstate: Programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Comput Commun Rev*. 2014;44(2):44-51.
8. Pontarelli S, Bonola M, Bianchi G, Capone A, Cascone C. Stateful openflow: Hardware proof of concept. In: IEEE High Performance Switching and Routing (HPSR); 2015:1-8.
9. Bianchi G, Bonola M, Pontarelli S, Sanvito D, Capone A, Cascone C. Open packet processor: a programmable architecture for wire speed platform-independent stateful in-network processing. *arXiv preprint arXiv:1605.01977*. 2016:1-14.
10. Moshref M, Bhargava A, Gupta A, Yu M, Govindan R. Flow-level state transition as a new switch primitive for SDN. In: 3rd Workshop on Hot Topics in Software Defined Networking; 2014:61-66.
11. The P4 language Consortium. The P4 Language Specification, version 1.0.2; 2015.
12. Bifulco R., Boite J, Bouet M, Schneider F. Improving sdn with inspired switches. In: Proceedings of the Symposium on SDN Research; 2016:11.
13. Jeyakumar V, Alizadeh M, Kim C, Mazieres D. Tiny packet programs for low-latency network control and monitoring. In: ACM Workshop on Hot Topics in Networks (HOTNETS 2013); 2013:8.
14. Gibb G, Varghese G, horowitz M, McKeown N. Design principles for packet parsers. In: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2013; 2013:13-24.
15. Intel Ethernet Switch FM6000 Series - Software Defined Networking. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf‘.
16. Waldvogel M, Varghese G, Turner J, Plattner B. Scalable high-speed prefix matching. *ACM Trans Comput Syst (TOCS)*. 2001;19(4):440-482.
17. Bosshart P. Low power TCAM US Patent 8,125,810; 2012.
18. Fedorov VV, Abusultan M, Khatri SP. An area-efficient ternary cam design using floating gate transistors. In: 2014 IEEE 32nd International Conference on Computer Design (ICCD); 2014:55-60.
19. Chang M-F, Lin C-C, Lee A, et al. A 3T1R nonvolatile TCAM using MLC ReRAM with Sub-1ns search time. In: 2015 IEEE International Solid-State Circuits Conference-(ISSCC); 2015:1-3.
20. Li J, Montoye RK, Ishii M, Chang L-Y. 1 Mb 0.41 μm^2 2T-2R Cell nonvolatile tcam with two-bit encoding and clocked self-referenced sensing. *IEEE J Solid State Circuits*. 2014;49(4):896-907.
21. Banakar R, Steinke S, Lee B-S, Balakrishnan M, Marwedel P. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In: CODES 2002, Proceedings of the Tenth International Symposium on Hardware/Software Codesign; 2002:73-78.
22. Hennessy JL, Patterson DA. *Computer architecture: A quantitative approach*. Elsevier; 2011.
23. Hennessy J, Jouppi N, Baskett F, Gill J. *MIPS: A VLSI Processor Architecture*. Berlin, Heidelberg: Springer; 1981.
24. Burger D, Austin TM, Bennett S. *Evaluating Future Microprocessors: The Simplescalar Tool Set*. Wisconsin, USA: University of Wisconsin-Madison, Computer Sciences Department; 1996.
25. MIPS CPU simulator. https://github.com/clord/MIPS-CPU-Simulator.
26. PMP CPU simulator project repository. https://github.com/marcobonola/PMP-simulator.
27. Zilberman N, Audzevich Y, Covington G, Moore A. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *Micro, IEEE*. 2014;34(5):32-41.
28. ONF. Openflow switch specification, version 1.4.0; 2013.
29. Jeyakumar V, Alizadeh M, Geng Y, Kim C, Mazières D. Millions of little minions: using packets for low latency network programming and visibility. *ACM SIGCOMM Comput Commun Rev*. 2015;44(4):3-14.
30. Dobrescu M, Egi N, Argyraki K, et al. Routebricks: Exploiting parallelism to scale software routers. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles; 2009:15-28.

31. Han S, Jang K, Park K, Moon S. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Comput Commun Rev*. 2011;41(4):195-206.
32. Zhou D, Fan B, Lim H, Kaminsky M, Andersen DG. Scalable, high performance ethernet forwarding with cuckooswitch. In: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies; 2013:97-108.
33. Cisco. quantumflow processor. http://newsroom.cisco.com/dlls/2008/hd_030408b.html.
34. Ezchip. np-5 network processor. http://www.ezchip.com/p_np5.htm.

Salvatore Pontarelli received the Master Degree at University of Bologna in 2000 and the PhD from University of Rome Tor Vergata in 2003. Currently, he is a senior researcher at CNIT (Italian Consortium of Telecommunications). Previously, he has worked with the National Research Council (CNR), the University of Rome Tor Vergata, the Italian Space Agency (ASI), and the University of Bristol. His research interests are high-speed packet processing and hardware architectures for software defined networks.

Marco Bonola received the PhD degree in telecommunications and microelectronics engineering in April 2011, from the University of Rome Tor Vergata. He has worked as a student intern at NTT DOCOMO EuroLab from 2007 to 2008. He has been involved in several EU projects: PERIMETER, DEMONS, OFELIA, BEBA, SUPERFLUIDITY, and 5G-PICTURE. His research interests are related to IP mobility, network security, traffic monitoring, and SDN. He is an author of the IETF RFC 6496. He is currently a senior researcher at CNIT (Italian Consortium of Telecommunications)

Giuseppe Bianchi has been a full professor of telecommunications at the School of Engineering of the University of Roma Tor Vergata since January 2007. He was formerly employed at Politecnico di Milano and the University of Palermo. He spent, in 1992, as a visitor researcher at the Washington University of St. Louis, Missouri, and in 1997, as a visitor researcher at the Columbia University of New York. His research activity (published about 170 papers in peer-refereed international journals and conferences) spans several areas, current active topics being wireless networks, network security, network measurement and monitoring, privacy. He has been involved in coordination roles (general, technical, or scientific coordinator) for three European STREP projects, one European IP project, and two national PRIN projects, and has participated as a unit coordinator to many other projects. He is an area editor of the IEEE Transactions on Wireless Communication, an editor for IEEE/ACM Transactions on Networking, and an area editor for Elsevier Computer Communications. He has chaired more than 10 international IEEE/ACM conferences or workshops. He is a senior member of the IEEE.

How to cite this article: Pontarelli S, Bonola M, Bianchi G. Smashing OpenFlow's "atomic" actions: programmable data plane packet manipulation in hardware. *Int J Network Mgmt*. 2018;e2043. https://doi.org/10.1002/nem.2043