Stateful Openflow: Hardware Proof of Concept

Salvatore Pontarelli* Marco Bonola*, Giuseppe Bianchi*, Antonio Capone†, Carmelo Cascone†,
CNIT/Universitá di Roma Tor Vergata †Politecnico di Milano

***Abstract*—This paper presents a hardware implementation of Openstate, an extension of OpenFlow that allows performing stateful control functionalities directly inside the switch, without requiring the intervention of an external controller. The paper shows how, with a minimal reworking of the OpenFlow's basic architecture, and reusing the same building blocks, it is possible to greatly extend the intelligence of an OpenFlow switch allowing the offload of many control task directly in the switch. An FPGA based implementation of an Openstate prototype is here presented, the different architectural design choices are discussed, and the performance and limitations of the developed prototype are examined. Finally, the paper proposes a discussion on the performance achievable by using an ASIC implementation of the OpenState switch[1].**

I. Introduction

Openflow [1] has gained in the last years significant attention from the networking research community. Openflow's technical contribution was a pragmatic platform agnostic interface for programing the forwarding behavior of network switches; its breakthrough innovation was the inspiration, and actual launch, of what we today refer to as Software Defined Networking [2].

Nevertheless, despite its game changing contribution, Openflow presents several functional limitations. The original OpenFlow's match/action (stateless) abstraction is indeed extremely effective in providing programmatic flexibility in the forwarding plane. However, the static nature of the forwarding rules inserted via OpenFlow inside the network switches requires to involve external (remote) controllers for any modifications in the forwarding plane, thus raising performance and latency concerns. Already in 2011 [3] SDN researchers fostered the necessity of enabling *state transitions at the data plane level* to "achieve ideal connectivity on a timescale much less than that of the control plane", which would be impossible to achieve with the "naive two planes (data vs control) approach".

Indeed, the question may be even more general: should control/data plane separation necessarily take the form of a *physical* separation, namely a "smart" controller (or network of controlling entities), which runs the control logic for "dumb" switching fabrics? The ability to *execute* inside single links/switches, at wire-speed, control tasks triggered by packet arrivals or local measurements, would bring about significant advantages in terms of reaction time and reduced signaling load towards external controllers. Notable cases of where local state and forwarding behavior updates would be by far better handled locally, do include, but not nearly limit to: layer 2 MAC learning updates, request-response matches in bidirectional flows, multi-flow sessions or protocols (e.g. FTP control on port 21 and data on port 20), NAT states, packet fragmentation states, and so on.

In 2014, in a work in progress paper [4], we proposed Openstate. To the best of our knowledge, [4] describes the first Openflow extension to support finite state machine execution at data plane. We focused on the architecture design and we proposed two use cases to show the benefit of the proposed solution. Nevertheless, even though we gave some hints on whether, and how, our proposed approach can be efficiently supported by existing OpenFlow hardware, in [4] we clearly stated that only an actual proof-of-concept HW implementation, which we did not have at that time, would have provided a compelling answer.

This paper fills this gap, and thoroughly addresses the *hardware feasibility of stateful handling and dynamic adaptation of forwarding rules*, while retaining an internal switch architecture and implementation as close as possible to the original OpenFlow's one. To back up such claim, this paper presents a minimal, but fully functional, FPGA based OpenState prototype, and discusses the (many) analogies and the (very few) extensions needed to adapt an OpenFlow hardware. Specifically, the paper makes the following contributions:

- we present a basic hardware architecture of OpenState, describing how the same basic building block of a typical Openflow switch can be re-used to perform stateful control task;
- we discuss and motivate the design choices made for implementing the few specifically needed OpenState extensions, as well as the relevant limitations, when applicable;
- using synthesis results, we show how the designed hardware fulfils the performance and design requirements;
- we finally discuss design and performance issues of a possible ASIC OpenState implementation.

The rest of this paper is organized as follows. In section II we briefly recall the Openstate abstraction and architecture design. In section III we describes the hardware implementations carried out using a FPGA prototype board. Sections IV presents the simulation and synthesis results for the development prototype and Section V discusses the limitations of the HW implementation and estimates the performance achievable with an ASIC implementation.

II. Overview of Openstate

In this section we briefly review the basic OpenState concepts, and provide the terminology and notation needed to understand the rest of the paper. The reader interested in more details and in a more extensive description may refer to [4].

[1]This work was partially supported by the European Commission in the frame of the BEBA project http://www.beba-project.eu/

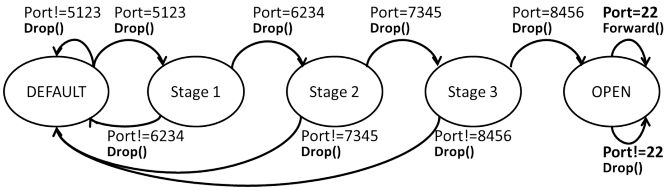

Fig. 1. port knocking example: Mealy (Finite State) Machine

A. From match/action to Mealy machines

In OpenFlow, a set of actions is associated to a flow match. OpenState exploits the *same* "match/action" primitive to add the stateful extension: the match is performed on packet header fields *plus a flow state label* (to be retrieved from a suitable State Table) and one of the actions associated to the match allows to update a flow state label. Note that a match not triggering any state transition (arguably the most common case) is readily accounted in OpenState under the special case of *self-transitions*, i.e. a transition from a state to itself.

The proposed approach can be formally modeled, in abstract form, by means of a simple *Mealy Machine*. We recall that a Mealy Machine is an abstract model comprising a 4-tuple $< S, I, O, T >$, plus an initial starting (default) state S0, where:

- S is a finite set of states;
- I is a finite set of input symbols (events);
- O is a finite set of output symbols (actions); and
- $T : S \times I \rightarrow S \times O$ is a transition function which maps <state, event> pairs into <next_state, action> pairs.

Similarly to the OpenFlow API, the abstraction is made concrete (while retaining platform independence) by restricting the set O of actions to those available in current OpenFlow devices, and by restricting the set I of events to OpenFlow matches on header fields and metadata easily implementable in hardware platforms. The finite set of states S (concretely, state labels, i.e., bit strings), and the relevant state transitions, in essence the "behavior" of a stateful application, are left to the programmer's freedom. As previously discussed, a transition function T is readily accommodated into a *single* TCAM entry, hence it uses the same OpenFlow hardware employed for ordinary match/action pairs.

B. Architecture overview

The OpenFlow data plane abstraction is based on a single table of match/action rules for version 1.0, and multiple tables from version 1.1 on. Unless *explicitly* changed by the remote controller through flow-mod messages, rules are *static*, i.e., all packets in a flow experience the same forwarding behavior. With OpenState, we introduce the notion of *stateful block*, as an extension of a *single* flow table. Stateful blocks can be pipelined with other stateful blocks as well as ordinary OpenFlow tables. A stateful block is an atomic block comprising two distinct, but interrelated, tables:

- a **State Table**, which stores the state labels associated to flow identities (no state stored meaning DEFAULT state), and
- an **FSM execution table**, which performs a (wildcard) match on a state label and the packet header fields, and returns an associated forwarding action (action set) and a next state label.

The programmer can specify the operation of a stateful block as follows:

- provide the list of entries to be loaded in the FSM table. Each entry in the FSM table comprises four columns: i) a **state** provided as a user-defined label, ii) an **event** expressed as an OpenFlow match, iii) a list of OpenFlow **actions**, and iv) a **next-state** label; each row is a designed state transition. At least one entry in the FSM table must use, in the first column, the DEFAULT state label;
- provide a "lookup-scope", namely the header field(s) of the packet which shall be used to access the state table during a lookup (read):
- provide a possibly different "update-scope", namely the header field(s) of the packet which shall be used to access the state table. We note that a straightforward extension could consist in what follows. Rather than using a *common* (unique) update scope for all the FSM entries, an extended implementation could permit the programmer to specify *different* update-scopes to be used as a parameter when calling for a state transition.

The decoupling between "lookup-scope" and "update-scope" permits to support a functional extension which we call *"cross-flow" state handling*. There are many practically useful stateful control tasks, in which states for a given flow are updated by events occurring on *different* flows. A prominent example is MAC learning: packets are forwarded using the *destination* MAC address, but the forwarding database is updated using the *source* MAC address. Similarly, the handling of bidirectional flows may encounter the same needs; for instance, the detection of a returning TCP SYNACK packet could trigger a state transition on the opposite direction.

OpenState offloads the control plane management from all the actions that can be defined inside the switch by using the above described programming model, reacting only to the few exceptions not covered by the defined Mealy machine. The main task of the control plane management of an OpenState switch mainly consists taking high level decisions that will be actuated programming the Openstate switch with a suitable FSM.

To best convey our concept, let's reuse (from [4]) a perhaps niche, but indeed very descriptive example: port knocking, a well-known method for opening a port on a firewall. An host IP that wants to establish a connection (say an SSH session, i.e., port 22) delivers a sequence of packets addressed to an ordered list of pre-specified closed ports, say ports 5123, 6234, 7345 and 8456. Once the exact sequence of packets is received, the firewall opens port 22 for the considered host. Before this stage, all packets (including the knocking ones) are dropped. This example can be easily implemented with the Mealy Machine illustrated in Fig. 1. Starting from a DEFAULT state, each correctly knocked port will cause a transition to a

series of three intermediate states, until a final OPEN state is reached. Any knock on a different port will reset the state to DEFAULT. When in the OPEN state, only packets addressed to port 22 will be forwarded; all remaining packets will be dropped, but without resetting the state. Note that a controller-based implementation of Port Knocking would require the switch to deliver *each and every packet received on a currently blocked port* to the controller itself!

III. Hardware Implementation

To gain understanding on the feasibility and wire-speed operation of OpenState, we have implemented a proof-of-concept hardware prototype using an experimental FPGA platform. The designed hardware prototype is conservative in terms of TCAM entries and clock frequency but it includes all the key OpenState components and features, including support for cross-flow state management. We remark that the main focus of this work is to show the feasibility of a hardware implementation, not to present a fully deployed Openstate FPGA implementation. With this scope, we don't focus on an efficient implementation of a well defined IP blocks such as TCAM, but we only use a simple implementation able to provide the TCAM functionality even if with a reduced number of entries. Further pushing our FPGA implementation is thus not only well out of the scope of this work, but it is also of limited practical relevance, as carrier-grade implementations in the order of Terabit/s throughput [5] in any case would require a custom ASIC implementation. Considering that the TCAM size does not affect throughput given its O(1) access time, we believe that even if at reduced scale, our implementation permits us to comparatively argue about complexity and performance with respect to an equivalent OpenFlow implementation.

A. Development platform

The OpenState hardware prototype has been designed using as target development board the INVEA COMBO-LXT [6], an express PCI x8 mother card equipped with the XILINX Virtex5 XC5VLX155T [7], two QDR RAM memories with a total capacity of 9MB and a throughput of 17166 Mbps for read and for write operations, and up to 4 GB of DDR2 memory. The 2 QDR II SRAM chips provide high bandwidth dual port memory for routing tables, flow memory, low latency data buffers. The board is equipped with a daughter board providing two 10 GbE interfaces, and is hosted in a PC workstation. The operating system sees the board as two Ethernet network cards connected using the express PCI bus. This allows to configure the development board as a 4 port switch in which two "virtual" ports are connected to the host workstation, while the other two ports can be connected to the external environment. All the four ports are connected to the FPGA that implements the OpenState architecture. The general scheme of the OpenState prototype is depicted in figure 2. The FPGA is clocked at 156.25 MHz, with a 64 bits data path from the Ethernet ports, corresponding to a 10 gbps throughput per port. Four *ingress queues* collect the packets coming from the ingress ports. A 4-input 1-output *mixer block* aggregates the packets using a round robin policy. The output of the mixer is a 320 bits data bus able to provide an overall throughput of 50 Gbps. A *delay queue* stores the packet during the time need by the OpenState tables to operate. The packets first go to the look-up and update extractor blocks that build the keys that are used to read/update the state table. The state table is composed by a d-left hash table, a TCAM and a companion SRAM. The new key, obtained composing the previous key and the extracted state is fed to the second TCAM/SRAM pair that is in charge of executing the FSM. The output of this TCAM/SRAM pair provides the command for the Action block and (if required) a new state that will be written in the hash table. In the following, a detailed description of the blocks composing the node prototype is presented.

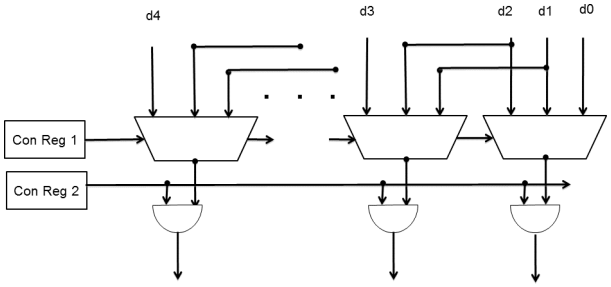

Fig. 3. hardware block performing the look-up/update extraction

B. Look-up/update extractors

The *look-up extractor* and the *update extractor* are implemented combining two basic operations: the first operation selects the beginning of the header ($i.e$ perform the shift of the input key), while the second operation is a 128 bits configurable mask operation. Two configuration registers have been used to configure the shifter and the mask. The barrel shifter used to select the beginning of the header, takes as input the initial 320 bits of the packet and provide as output 128 contiguous bits starting from the ith bit defined by the first configuration register. For this configuration register two bytes are more than sufficient, since correspond to a interval value for the index between 0 and 1023. The mask operation is simply the bitwise 'and' between the output of the barrel shifter and the second configuration mask register. Since each bit can be individually masked, this configuration register requires 9 bytes. Figure 3 presents the hardware block that performs the look-up/update extraction.

The logical operations composing the extractor are easily implementable in hardware and are able to process the incoming data at the required clock frequency, while maintaining a high degree of flexibility needed to select different kind of protocol fields. If the case of the MAC learning example, the lookup scope, that is the destination MAC address, can be configured setting the shift configuration register to 0 and the mask configuration register to mask all but the first 48 bits of the packet. Instead, for the update scope, we can set the shift

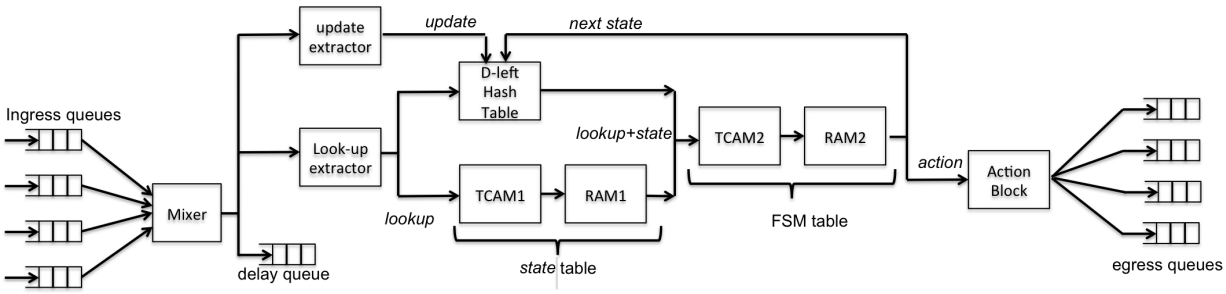

Fig. 2. Scheme of the OpenState hardware prototype

configuration register to 48 and the same mask configuration register used for the destination MAC.

We note that in this case, we have multiple choices for the selection of the source MAC, since both the source MAC and the destination MAC are very close, and therefore the selection of the source or destination MAC address can also be done only changing the mask configuration register. Instead, if the look-up or update scope refers to non contiguous header fields, the use of the index allows the selection of these fields. We remark that the use of a configuration mask allows to combine together multiple fields to form the scope ($e.g.$ can be combined the field of the source IP and the field of the destination port of a TCP packet), with the limitation that the bits composing these fields are in the same 128 bits wide windows defined by the index configuration register.

This limitation does not seems particularly important, since typical scopes can refer to multiple fields, but they are usually contiguous fields, or fields with a limited distance. Instead, the use of two different pairs of registers for the lookup and update scope selection allows to decoupling the header fields used for the state lookup form the header fields used for the update.

Another motivation that drives the selection of this index/mask implementation of the extractor block is that the extractors always provide as output a fixed width 128 bits value (even if often some of the 128 are always masked to 0). This allows to directly use the key provided from the extractors as input for the TCAM and the hash table used to implement the state table.

Furthermore, this implementation of the extractors allows to easily extend the Openstate functionality to select the update scope depending on the state of the packet under inspection. In fact, the per-state update scope selection only requires to store the value of the 2 registers (the 2 bytes of the index register plus the 9 bytes of the mask register) in the FSM execution table, and to apply these values to configure the update scope extractor block.

As a last remark, we notice that the lookup/update extractor configuration is somewhat similar to the format of Protocol-Oblivious Forwarding (POF) element [8]. While the lookup/update extractor configuration is composed by an offset and mask, the corresponding POF element is formed by an offset and a length. However, the mask configuration register can be used to shorten the length of the key to extract,($i.e.$ setting to 0 the last n bits of the incoming input). Therefore this simple block can provide a superset of the operations provided by a POF element.

C. State table

The state table provides the state associated to a flow identified by the look-up extractor. The actual implementation of the state table will depends on several design parameters (hardware or software, type of memories, latency, throughput, number of flows to manage etc). For the hardware implementation a good compromize between flexibility and hardware cost consists in implementing the state table by using a hash table and a small TCAM. In our proof-of-concept FPGA implementation the TCAM has 32 entries of 128 bits (TCAM1), and an associated Block RAM of 32 entries of 32 bits (RAM1) that reads the output of the TCAM and provides the state associated to the specific TCAM row. As already anticipated, the limited size of the TCAM is due to the difficulties of implementing efficient TCAMs using the FPGA resources. We outline that an effective and scalable FPGA implementation of Ternary Content Addressable Memories is a widely open research issue [9], [10], [11], especially since the priority resolution hardware limits the maximum operating frequency when the number of TCAM entries increase. Indeed, the achievable performance with FPGA TCAM are still far, in terms of size and clock frequency, from those attainable by a full custom ASIC TCAM design [12]. The TCAM is needed to handle special (wildcard) cases, such as static state assignment to a pool of flows (e.g. ACLs), flow categories which are out of the scope of the machine and must be processed in a different way (if necessary by another stage, either stateless or stateful). Instead, the hash table keep track of the state of the flows traveling in the network. The hash table is realized by using a *d-left hash table*, with d=4. The table is sized for 4K entries, and is accessed with 128 bit keys provided, alternatively, by the *look-up extractor* or the *update extractor* during state read and (re)write, respectively. The choice of the right hash table structure to realize the state table is one of the most important design choice, since this impact on the scalability of the solution (since this hash table will store the state for all the individual flows active in the network) and on the maximum throughput (since for each packet a look-up on the hash table must be performed). The requirement of at look-up for each

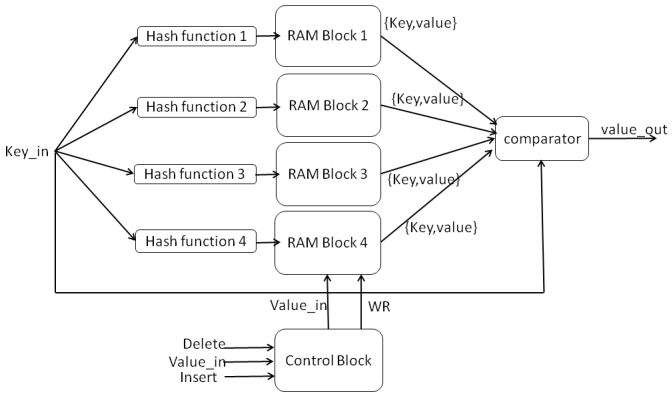

Fig. 4. hardware block of the d-left hash table

packet make requires an hash table with a constant access time. This choice avoids the issues related to a non fixed worst case delay, that could compromise the overall performance of the system. Therefore, we implemented the hash table using a multiple hash structure (MHT) such as the ones described in [13]. Since the different hash tables can be accessed in parallel, the look-up of a key can be performed in only one clock cycle, avoiding that the access to the hash table become an bottleneck for the system. However, in order to simplify the hardware structure, and to avoid a bottleneck in the update of the state, also the insertion of items in the hash table must be performed in a fixed number (namely one) of clock cycles. This require to avoid the use of MHT with moving capabilities (such as the well-know cuckoo hash tables) since the insertion time of these structure is not deterministic. This choice, that correspond to the implementation of a d-left hash table, will also bring a big simplification in the management of the multiple hash table. The drawback of this choice is the reduced memory efficiency with respect to the cuckoo hashing. While a 4-way cuckoo hash can reach the 99% memory occupancy, the d-left hash permit a memory occupancy of the order of 70%.

Figure 4 shows the blocks composing the d-left hash table. The four hash blocks perform four different H3 hash functions [14]. These hash functions has been chosen since they can be easily implemented in hardware. The four RAM blocks store the content of the hash table, while the comparator block checks which RAM block actually stores the queried key and provides the value associated to that key. In order to provide a non-blocking behavior of the hash table, the FPGA Block RAMs are configured to work as dual port RAMs, with one write port and one read port. This choice permits to update the status of a flow without blocking the lookup of another state[2]. Finally, the control block manages the insert,remove and write signals needed to update/delete the hash table entries.

D. FSM table

As for the state table, also the FSM execution table is an abstract structure that can be implemented in several ways depending on the underlying hardware that will provide the FSM execution functionalities. In our prototype the FSM execution table is implemented by a TCAM. This choice allows the highest degree of flexibility while providing a constant one clock cycle to execute the FSM transition. The TCAM has 128 entries of 160 bits (TCAM2), associated to a Block RAM of 128 entries of 64 bits (RAM2), storing the next state used to update the state table, and the specific action to perform on the packet. This TCAM takes as input the aggregated lookup-scope and the retrieved flow state, providing as output row associated to the matching rule with higher priority. As previously mentioned, the limited number of entries of the TCAMs is due to the inefficient mapping of these structures on an FPGA. This number, however, is similar to that of other FPGA based TCAM implementations, such as [15].

[2]if the same entry of the RAM is concurrently read and written, the block RAM is configured to provide the new value to the read port.

E. Packet output

A final *Action Block* applies the retrieved action to the packet coming from the delay queue. Being our prototype a proof-of-concept, as of now only a basic subset of OpenFlow actions have been implemented: drop, select (enable one or more of the output ports to forward the packet), and tag (insert/modify/remove the VLAN tag). This block then provides as output the four 64 bits data-bus for the four 10 Gbits/sec egress ports. The Action block is realized composing several blocks that provide elementary operations. All the blocks share the same I/O interface, that is composed by a input port and one of more egress ports. This choice will permit to enhance the functionalities of the prototype adding new blocks if it is needed to perform more actions. For the implemented prototype the first elementary block allows to insert/remove the VLAN tag, or to modify the VLAN tag value. The subsequent elementary block is a select block that takes as input the packet and provide one output for each port. Depending on the value of the action given by the FSM execution table the block selects on which output ports the packet must be forwarded. If no output are selected, the packet is dropped.

IV. Simulation & synthesis results

In order to show the behavior of the Openstate prototype, in this subsection we presented a small simulation of Openstate. The system has been configured to implement the port-knocking example that has been previously described. The system inspects several packets, changing the state corresponding to the SRC IP of the incoming requests and dropping the packet until the final state in reached. Figure 5 presents the result of the simulation.

The waveform shows the ingress bus (only one of the ingress queues is presented in the waveform), the four egress queues, and some signals of the hash table and of the TCAM1 and TCAM2 blocks. In particular, it is possible to see that the first packet only corresponds to a match in the TCAM1, (the flow is in the default state), while the subsequent packets coming from the same SRC IP, are matched also by the hash table. The signals of TCAM2 shows the state transitions that occur during the processing of the packets. The flow state starts

Fig. 5. Captured waveform of the Openstate prototype simulation

from the DEFAULT state, (labeled as 16), and moves to the intermediate states (labeled as 11,10,9) while the packets with the right TCP destination port arrives. At the end of the state transitions, the flow state arrives in the OPEN state (labeled as 5), which corresponds to the forwarding action (the value of the action signal is 0x00000020) in which the packets are transmitted of out the port tx2 of the switch. Instead, during the intermediate states the packet are dropped, as indicated by the action signal (the value of the action signal is 0x000000100), which set the drop flag for the action block.

The whole system has been synthesized using the standard Xilinx design flow: the resource occupation for the implemented system, in terms of used logic resources, are presented in the table below.

type of resource	# of used resources	[%]
Number of Slice LUTs	10,691 out of 24,320	43%
Block RAMs	53 out of 212	25 %

TABLE I
SYNTHESIS RESULTS

The Openstate prototype requires less that half the logic resources of the FPGA. This result proves that the Openstate architecture is well suited for an hardware implementation, since the implemented system is a able to provide a minimal but complete implementation of the Openstate concept, and is able to sustain a considerable throughput. This is particularly evident considering also that the used FPGA is quite obsolete now (it is realized with a 65 nm manufacturing process, while the current state of the art FPGA are realized with a 28nm process), both in terms of logic resources and of maximum frequency.

V. Discussion, Limitations, Extensions

The FPGA prototype confirms the feasibility of the OpenState implementation. The additional hardware needed to support cross-flow state management (namely, the extractor modules) uses a negligible amount of logic resources and does not exhibit any implementation criticality. Similarly, the limited number of actions and TCAM entries implemented in the prototype are just due to the proof-of-concept nature of our prototype (and lack of an OpenFlow hardware from which OpenState would directly inherit these parts).

A. Limitations

If compared with an OpenFlow implementation, OpenState exhibits only one (minor) shortcoming. The system latency, i.e. the time interval from the first table lookup to the last state update is 5 clock cycles. The FPGA prototype is able to sustain the full throughput of 40 Gbits/sec provided by the 4 switch ports. If we suppose a minimum packet size of 40 bytes (320 bits), the system is able to process 1 packet for each clock cycle, and thus up to 5 packets could be pipelined. However, the feedback loop (not present in the forward-only OpenFlow pipelines [16]) raises a concern: the state update performed for a packet at the fifth clock cycle would be missed by pipelined packets. This could be an issue for packets *belonging to a same flow* arriving back-to-back (consecutive clock cycles); in practice, as long as the system is configured to work by aggregating $N \geq 5$ different links, the mixer's round robin policy will separate two packets coming from the same link of N clock cycles, thus solving the problem. Note that the 5 clock cycles latency is fixed by the hardware blocks used in the FPGA (the TCAM and the Block RAMs) and basically does not change scaling up the number of ingress ports or moving to an ASIC. Moreover, we remark that also the typical control update mechanism of OpenFlow does not allow to exactly determinate at which time instant a new rule is installed in the flow tables, since this update is heavily dependent on the way in which the table are implemented in the actual OpenFlow switch.

B. Performance achievable with an ASIC implementation

As previously stated, while an FPGA prototype permits to assess feasibility, a full performance/scale architecture requires ASIC technology. Following the same technology assumptions of [5], an OpenState ASIC design would be able to work at 1GHz operating frequency. This corresponds to an aggregate throughput of 960M packets/s, that is the maximum achievable

by a 64 ports 10 Gb/s switch chip. However, the most important scaling provided by the ASIC implementation is given by the number of entries that can be stored in the OpenState tables. The size of the SRAM that can be instantiated on a last generation chip is up to 32 MB, corresponding to 2 millions of entries in the d-left hash for the Flow table. The size of a TCAM can be up to 40 Mb, corresponding to 256K FSM table entries.

VI. Related work

OpenFlow, now at version 1.4 [16], has undergone several extensions (more flexible header matching, action bundles, pipelined tables, synchronized tables, multiple controllers, and many more), and new important ones are currently under discussion, including typed tables [17] and flow states [18]. The need to rethink OpenFlow data plane abstraction has been recently recognized by the research community [5], [19], [8]. In [5], the authors point out that the rigid table structure of current hardware switches limits the flexibility of OpenFlow packet processing to matching on a fixed set of fields and to a small set of actions, and propose a logical table structure, RMT (Reconfigurable Match Table), on top of the existing fixed physical tables and new action primitives. Notably, the proposed scheme allows not only to consider arbitrary width and depth of the matching for the header vector but also to define actions that can take input arguments and rewrite header fields. Along the same line but with a more general approach, in [19] the authors propose P4, a high-level language to program packet processors which focus on protocol-independence. P4 allows the programmer to define packet parser (able to support matching on new header fields) and to implement arbitrary parametric actions as the composition of reusable primitives for packet header manipulation. It must be said that P4 is presented as a straw-man proposal for how OpenFlow should evolve in the future, rather than a proof-of-concept. In [8] a Protocol-Oblivious Forwarding (POF) abstraction model is proposed as a set of low-level Forwarding Instruction Set (FIS) (comparable to those found in the assembly language). Especially, the authors propose new stateful instructions to actively manipulate the flow table entries. Even though prototype implementations (hardware and software-based) are presented, POF can be considered a clean slate proposal with substantial differences from existing SDN programming abstractions and implementations that can hardly be put into an evolutionary perspective. In [20], the approach is even more radical and, similarly to the early work on active networks, packets are allowed to carry a tiny code that define processing in the switch data plane. A very interesting aspect is the proposal of targeted ASIC implementations where an extremely small set of instructions and memory space can be used to define packet processing. Finally, also an FSM model is at the basis of the approach to define network policies in PyResonance [21]. However, FSMs in PyResonance are defined at the controller level and later translated to OpenFlow rules using a reactive approach.

VII. Conclusions

OpenState is a first attempt to permit platform-agnostic programming of a subset of stateful control functions directly inside the switches, while retaining wire-speed performance. In this paper we have shown the feasibility of an hardware implementation of Openstate, we have described the main blocks composing the prototype and we have discussed the performance obtained in the current FPGA prototype, evaluating the performance achievable upgrading in case of an ASIC implementation.

In particular, we showed that the stateful extension can be developed reusing the same building block of an standard OpenFlow implementation, while the proposed extensions (mainly the support for "cross-flow" state handling, i.e. permit the arrival of a packet of a given flow to trigger a state transition for a different flow) can be implemented with simple yet efficient combinatorial hardware blocks.

The proposed implementation is highly scalable, since the main issue in terms of scalability is related to the number of flow states (that are not in the DEFAULT state) to manage. Since these states can be easily stored in a d-left hash table, the scalability of the system is related to the maximum size of the SRAM memory (more than 2 millions of flows can be stored in a 32 MB embedded SRAM).

Finally, from the implementation we did, we learned that one of the expected issues, $i.e.$ number of clock cycles required to update a state, can be limited to few clock cycles, thus allowing a state update so fast that do not compromise the functionality of OpenState even when back to back packets belonging to the same flow and coming from the same network interface should be analyzed.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[2] K. Greene, "TR10: Software-defined networking, 2009," MIT Technology Review, available online at http://www2.technologyreview.com/article/412194/tr10-software-defined-networking.

[3] J. Liu, B. Yan, S. Shenker, and M. Schapira, "Data-driven network connectivity," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM, 2011, p. 8.

[4] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[5] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM 2013*. ACM, 2013, pp. 99–110.

[6] "COMBO Product Brief," http://www.invea-tech.com/data/combo/combo_pb_en.pdf.

[7] "Virtex-5 Family Overview," http://www.xilinx.com/.

[8] H. Song, "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*, ser. HotSDN '13. ACM, 2013, pp. 127–132.

[9] B. Jean-Louis, "Using block RAM for high performance read/write TCAMs," 2012.

[10] Z. Ullah, M. Jaiswal, Y. Chan, and R. Cheung, "FPGA Implementation of SRAM-based Ternary Content Addressable Memory," in *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2012.

[11] W. Jiang, "Scalable ternary content addressable memory implementation using FPGAs," in *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*. IEEE, 2013, pp. 71–82.

[12] P. K. and A. S., "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[13] A. Kirsch, M. Mitzenmacher, and G. Varghese, "Hash-based techniques for high-speed packet processing," in *Algorithms for Next Generation Networks*. Springer, 2010, pp. 181–218.

[14] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *Computers, IEEE Transactions on*, vol. 46, no. 12, pp. 1378–1381, Dec 1997.

[15] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow switch on the NetFPGA platform," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2008, pp. 1–9.

[16] Open Networking Foundation, "OpenFlow Switch Specification ver 1.4," Tech. Rep., Oct. 2013.

[17] http: //goo.gl/TtLtw0, "Openflow forwarding abstractions working group charter," in *Apr. 2013*.

[18] B. Mack-Crane, "Openflow extensions," in *US Ignite ONF GENI workshop, October 8, 2013*.

[19] P. Bosshart, D. Daly, M. Izzard, N. McKeown, J. Rexford, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "Programming protocol-independent packet processors," *CoRR*, vol. abs/1312.1719, 2013.

[20] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazieres, "Tiny packet programs for low-latency network control and monitoring," in *ACM Workshop on Hot Topics in Networks (HOTNETS 2013)*, 2013.

[21] "PyResonance," https://github.com/Resonance-SDN/pyresonance/wiki.