A 100Gig Network Processor platform for Openflow

Omar El Ferkouss*, Ilyas Snaiki*, Omar Mounaouar*, Hamza Dahmouni‡,
Racha Ben Ali†, Yves Lemieux†,Cherkaoui Omar*
*Université du Québec à Montréal (UQAM)
{elferkouss.omar, ilyas.snaiki, mounaouar, cherkaoui.omar}@netvirt.ca
‡Institut national des postes et télécommunication (INPT)
{dahmouni}@inpt.ac.ma
†Ericsson Research Canada
{racha.ben.ali, yves.lemieux}@ericsson.com

***Abstract*—Openflow splits the control plane from the data plane and move it to a centralized remote controller that dictates the forwarding behaviour to multiple Openflow switches. OpenFlow 1.1, introduces pipeline processing of the flow tables. This helps to improve flexibility and performance. This paper presents an implementation of the version 1.1 of OpenFlow capable to achieve a good performance. Knowing that OpenFlow is a rule-based approach, we can have several levels of flexibility of the switch by only updating the flow entries on-the-fly. We will show also that we can achieve a good performance using RFC (Recursive Flow Classification) that uses SRAM and TCAM side by side to enhance classification performance for the Openflow 1.1 Switch. Using this approach helps ameliorating the parallelism of the classification process and better exploiting of the hardware resources since most hardware platforms (such as Network processors) contain both TCAM and SRAM to store the lookup structures.**

I. Introduction

Openflow [1] is a new practical approach in networking that is gaining popularity among researchers both in academia and industry. This is due partly to its flexibility in quickly deploying innovative networking applications, inline with the software defined network vision, without going through the painful and slow standardization process of new protocols. This Openflow approach splits the control plane from the data plane and move it to a logically centralized remote controller. This controller maintains all the networking logic in different software applications that push forwarding and packet processing rules to the data plane either proactively or reactively. The genericity of Openflow is achieved by omitting the separation between layers and performing packet processing on arbitrary packet header fields in any combination of layers simultaneously. 14 tuples are considered for the latest version 1.1 of Openflow protocol. However performing packet header lookups on up to 14 tuples at once is considered among the biggest challenges of Openflow that prevent it from scaling to networks larger than campus networks and data center networks.

Some of the interesting research in the literature, such as DIFANE [2], has already tackled this data plane scalability issue of Openflow by partitioning the rule space into different classes of rules depending on their level of granularity or specificity. Then different roles are assigned to different switches depending on the classes of rules that each one maintains. With this DIFANE approach the lookup is distributed and balanced between these switches without leaving the data plane, i.e without the need to invoke the centralized controller for the first packet of each new unmatched flow.

In our proposed network processor based approach, pushing Openflow classifier wildcard rules in TCAM hardware memory accelerates the lookup on multiple fields within a few number of clock cycles. However, the TCAM is a scarce, expensive and energy-consuming memory with a limited number of entries compared to the number of possible 14-tuples rules that an application may want to push. Therefore, Openflow classifier specific rules and 14-tuples exact match rules have to be pushed into other large-capacity memories such as SRAMs. SRAMs can be external to network processors to allow much more larger memory capacity. However, external large-capacity memories usually have an access time worse than internal small-capacity memories, and may therefore affect the Openflow packet lookup performance. Furthermore, search structures in these memories such as tries for wildcard rules and hashes (allowing collisions) for exact match rules usually have a slower lookup than TCAM memory.

In order to reduce the use of the TCAM without invoking the centralized controller for each new 14-tuples micro flow, the DevoFlow approach in [3] automatically clones a matched general wildcard rule in the TCAM by adding a specific exact match rule in the SRAM. Therefore, the exact match 14 tuples flow granularity of Openflow is maintained, which is required for per micro flow specific QoS actions for instance, without leaving the data plane, i.e. the fast path.

In order to accelerate the general packet processing, Openflow 1.1 introduces multiple lookup tables to allow a pipeline processing. Therefore, in our paper, we identified different implementation designs of Openflow 1.1 multiple tables over network processors that we are currently evaluating. These multiple tables are mapped to different types of memories depending on network application requirements on hardware performance vs. behavior change flexibility.

II. OpenFlow Classification challenges

In an OpenFlow network, traffic is classified based on the flow descriptor built from L2, L3 and L4 fields of each packet, called match fields. The classification processing time

increases as the match fields length increase, which means in an OpenFlow 1.1 network that uses 14 tuples as match fields, the classification process requires more time to retrieve the decision. This time requirement reduces significantly the performance of an OpenFlow switch. By looking to the available classification techniques [4], we find that the RFC (Recursive Flow Classification) algorithm may be the appropriate one to be used since the classification process in the OpenFlow 1.1 context is a multi-dimensional problem. It can help to increase the lookup time performance since it can perform parallel lookups through pipelined stages. Later in this paper, we propose a new approach based on the RFC algorithm, called Extended RFC, that can help us enhancing the performance of the OpenFlow 1.1 switch by using both TCAM and SRAM simultaneously.

III. OpenFlow switch over Network processor

The architecture of our implementation of OpenFlow 1.1 [5] over a Pizza Box has 2 main parts: the data plane and control plane part. On the data plane the packets are processed by the Network Processor which merely enforces the actions added on the control plane. The control plane on other hand has basically the OpenFlow Preprocessor Engine. It receives the configurations from the NOX [6] OpenFlow controller and transforms them in entries for the tables that are accessed by the network processor.

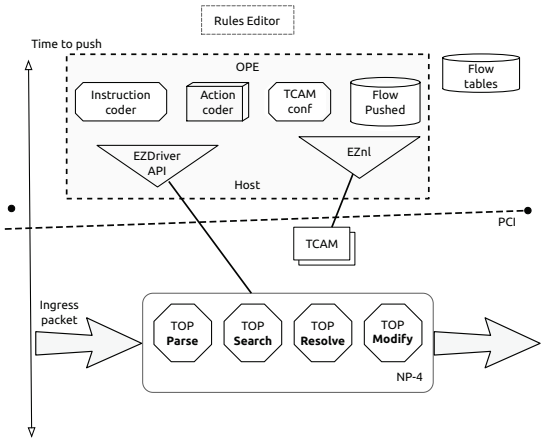


Fig. 1. Architecture of the OPE

The OPE (OpenFlow Preprocessor Engine) is responsible for managing the Network Processor (NP) and for intermediating the communication between the NP and NOX. It is responsible for pushing the entries and policies received from NOX into the NP. Its architecture is shown in figure 1.

The EZchip NP4 [7] is a Network Processor with mixed pipeline and parallel architecture (figure 2) capable of achieving speeds of 100Gbps. The EZchip NP4 has 4 different Task Optimized Processors. They are organized in a pipeline fashion: when a packet is being processed by the second TOP, another packet is already being processed by the first. This, combined with the parallelism, can make the processor achieve high network throughput.

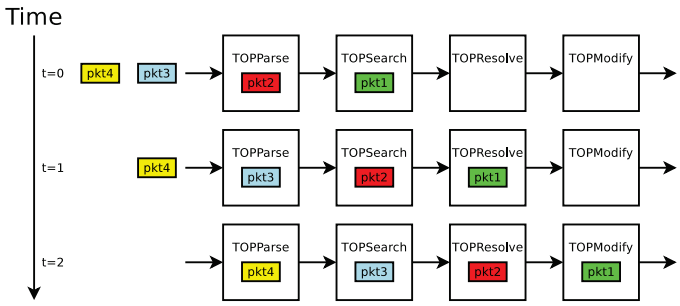


Fig. 2. EZchip pipeline

IV. Implementation

A. Design issues

The challenges behind an OpenFlow switch is to provide the flexibility and performance of an OpenFlow switch with different level of granularities. Indeed, the issues involved here are:

Flexibility: one of the main requirements of the system is the ability to easily change the NP programming without having to reload any new code, but just with changes in the flow tables. This is the flexibility in terms of flow entries. The OpenFlow table pipeline introduced with the 1.1 version seems suitable to model this interaction. The flexibility is evaluated by the time required by the software to change the hardware behaviour. It is the time to push the entries in the flow tables. A smaller time of update of the flow tables ensures the best flexibility of the system. We have also the flexibility in terms of applications. It means the ability to add a new application or change the application behaviour on-the-fly.

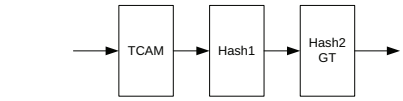
Performance: to take full advantage of the OpenFlow table pipeline aforementioned, the implementation should match the OpenFlow pipeline with the NP pipeline, in order to efficiently use the processing power offered by the NP – or more specifically its task-optimized processors (TOPs). Performance is a key paradigm because we want to be able to scale *while being flexible*.

Resource management: the implementation is also required to manage well the available resources, looking into minimizing expensive operations (e.g. the lookups) and also to use scarce resources (e.g. the TCAM) in an efficient way, especially because the TCAM will be used to dictate the router behavior.

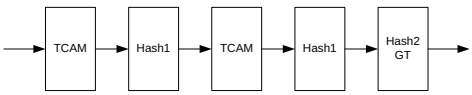
B. Different Implementations

This sections explains several design choices that we made to implement the version 1.1 of OpenFlow over EZchip NP-4. The first design, which is $I0$ is shown in figure 3(a) and already presented in the latest GENI Conference [8], uses a TCAM and a hash table in the first level. The match fields are in the TCAM while the first hash table is used to store the TCAM indexes. In the second level, we can have several group tables stored in EZchip hash tables to contain the actions of each application.

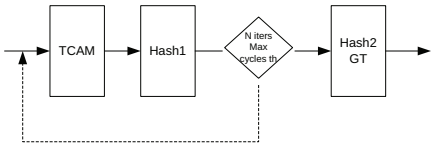
The second design, the implementation $I1$ shown in figure 3(b), is an extension of $I0$ that uses two level of TCAM in order to distribute the match fields. It means that if we have more than one field used for the matching, it would be better to distribute them in several tables. Thus, the $I1$ uses one stage of TCAM and hash table more than the implementation $I0$.



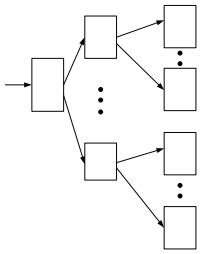
(a) $I0$ implementation



(b) $I1$ implementation



(c) $I2$ implementation



(d) $I3$ implementation

Fig. 3. Different implementations

The third implementation, which is shown in figure 3(c), is a generalization of $I2$ that can use more than 2 levels of TCAM. It can be very useful when we use more than 2 fields to match the packet. Note that the number of iterations N must be chosen so that the performance of the switch will not be affected.

All the implementations above use the TCAM to match the used fields and wildcard the other OpenFlow fields. We propose also the design shown in figure 3(d). The idea is that the parsing of the packet will be by stages instead of parsing the whole packet header at the beginning. Thus, the result of each step indicates the next field(s) and the tables that will be used for lookup.

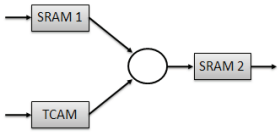
C. Extended Recurcive Flow Classification

The recursive flow classification [9] is an SRAM-based algorithm. Since a classifier contains usually both wildcarded and exact value bits, we try to propose a new way to combine SRAM and TCAM in the classification process in order to get better performance. This approach tends to reorder the classifier and distribute it over the TCAM and the SRAM to benefit from the strength of each one, which is the wildcard match in TCAM and the exact match in the SRAM. Also, most of hardware platforms, such as Network Processors and NetFPGA [10], contain both SRAM and TCAM memories. This approach can be a good way to better exploit the hardware resources. In the following, we describe the pre-processing and the processing of our extended RFC approach.

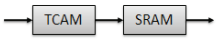
a) Pre-processing: The ExRFC (Extended RFC) pre-processing consists of preparing the data structure to be used for the classification process. It is done over three phases: *Interleaving*, *Splitting* and *RFC pre-processing*.

The first phase, i.e. *Interleaving*, consists of reordering the original classifier columns, each one is a 1-bit width, in order to give the best group of wildcarded bits and exact bits. The interleaving order of columns has to be maintained to be applied on the manipulated header used for the classification. In the second phase, i.e. *Splitting*, the interleaved classifier is divided into two parts. The first part is SRAM-based. The second part is TCAM-based. In the last phase, i.e. *RFC pre-processing*, the data structure is built in accordance with the splitting. The SRAM-based part will be pre-processed using the classic RFC algorithm. As for the TCAM-based part, it will be kept as it is, after eliminating redundancies.

b) Processing: When a packet arrives, the relevant fields are extracted to build the header used for the classification. This header is interleaved using the order applied on the original classifier and split into two chunks. The first chunk will be used to be processed using the TCAM. The second chunk is split again into chunks to be used by the RFC processing. The processing is either done through a parallel architecture, as shown in figure 4(a), which may offer better performance , or a pipeline architecture, as shown in figure 4(b), which may offer better flexibility.



(a) Parallel Architecture



(b) Pipeline Architecture

Fig. 4. Parallel vs Pipeline architecture

V. Results

A. Flexibility

Table I shows some numerical results: the time to update an entry on the TCAM and the Hash Table (which need to be updated every time you add an entry) and the total time to add a flow entry for the implementation $I0$, $I1$ and $I2$.

Concerning the flexibility in term of adding a new application, or changing the application behaviour, the implementation $I3$ is the best one. Indeed, for example for the routing application, the used match fields are the destination MAC

address and the IP destination address . So, using $I0$, the TCAM entry uses two fields and therefore we have to take into account all possible combinations. On the other hand, using $I2$ or $I3$ we can use two TCAM tables and save memory space. Table II shows some numerical results for the routing application that uses 16 destination MAC addresses, 400 prefixes and 200 next hops. Note that if we have more than 3 match fields, $I3$ is the best implementation in term of the flexibility for adding or changing application behaviour.

Table III present a comparison between the different design choices. As show in the table, $I0$ is the most flexible implementation in term of adding entries for a single flow since it required just 3 entries. But for applications, $I0$ might not be a good idea especially if the application uses several match fields. In this case, $I3$ shown in figure 3(d) is the best one since we can reserve a stage for application. Indeed, the first stage indicates the types of the application (we can have several applications running in the same switch) while the following stages are for the processing of each application. Also, since we do not parse the whole packet to extract the OpenFlow 1.1 14 tuples, $I3$ can achieve a good performance comparing to the other implementations.

B. Performance

Table IV shows performance results (i.e packet processing time in EZchip clock cycles) of the Openflow 1.1 Switch Implementations $I0$ and $I1$ assuming that there is no bottleneck in the lookup level or memory access problems. The best case measurement is based on unresolved packets, and the worst case is calculated based on packets that requires VLAN and IPv4 routing processing. Note that the EZchip NP-4 system clock is 400MHz and therefore each clock cycle takes $2.5ns$. Take also into account that the EZchip NP-4 embed 32 parallel engines at each TOP level.

Memory Type	Update Time (1 entry)
TCAM	450us
Hash Table	875us
Total:I0	2.2ms
Total:I1	3.5ms
Total:I2	6.17ms

TABLE I
TIME FOR ADDING ENTRIES

Implementation	Update Time
I0	8 655ms
I1	726.2ms
I2 (N=2)	726.2ms

TABLE II
TIME FOR ADDING ENTRIES FOR ROUTING APPLICATION

VI. CONCLUSION

In this short paper, we present our working version of Openflow 1.1 multiple tables pipelining over 100Gbps hybrid network processors. We describe several designs of chaining these pipelined lookup tables and mapping them to different types of memories, essentially TCAMs and SRAMs.

	I0	I1	I2	I3
Entries flexibility	1	2	3	1-3
Add new applications	3-4	3	2	1
Change Applications Behavior	1-3	2-4	3	1
Memory complexity	3	2	1	1-2
Lookup performance	1-2	2-3	3-4	1
Memory consumption	4-5	4	3	1
Memory adjustment	3	2	1	1

TABLE III
COMPARING DESIGN CHOICES (1:GOOD, ..., 5:BAD)

	I0	I1
Clock cycles in best case	224	291
Clock cycles in worst case	378	399

TABLE IV
IMPLEMENTATION $I0$ AND $I1$ PERFORMANCES

Depending on network application requirements in terms of lookup performance vs. update flexibility tradeoff, a design can be more suitable than the other. Therefore, the identified designs are qualitatively compared based on these requirements. We also present preliminary results regarding the update time as a flexiblity measure and the micro-codes clock cycles as a processing performance measure. As a future work, we planned exhaustive performance evaluation of our implementations at line rate speeds using IXIA hardware traffic generator.

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 Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] M. Yu, J. Rexford, M. Freedman, and J. Wang, "Scalable flow-based networking with difane," in *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*. ACM, 2010, pp. 351–362.

[3] J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. Curtis, and S. Banerjee, "Devoflow: Cost-effective flow management for high performance enterprise networks," 2010.

[4] P. Gupta and N. McKeown, "Algorithms for packet classification," *Network, IEEE*, vol. 15, no. 2, pp. 24–32, 2001.

[5] "Openflow switch specification version 1.1.0 - draft 4," Feb 2011.

[6] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[7] E. Technologies, "NP-4 100-Gigabit Network Processor for Carrier Ethernet Applications Product Brief," Tech. Rep., 2011.

[8] O. El Ferkouss, R. Matela, S. Correia, B. Boughzala, Y. Lemieux, R. Ben Ali, M. Tatipamula, M. Lemay, O. Cherkaoui, "A 100 Gbps Openflow 1.1 Switch," Tech. Rep., 2011. [Online]. Available: http://groups.geni.net/geni/attachment/wiki/GEC10DemoSummary/UQAM-Openflow-Genidemo-Poster_V_1.0.pdf

[9] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 1999, pp. 147–160.

[10] G. Watson, N. McKeown, and M. Casado, "Netfpga: A tool for network research and education," in *Workshop on Architecture Research using FPGA Platforms*. Citeseer, 2006.