International Journal of Electrical and Computer Engineering (IJECE)
Vol. 8, No. 5, October 2018, pp. 3021 – 3037
ISSN: 2088-8708, DOI: 10.11591/ijece.v8i5.pp3021-3037

Network Function Modeling and Performance Estimation

Mario Baldi and Amedeo Sapio
Department of Control and Computer Engineering, Politecnico di Torino, Italy

Article Info

Article history:
Received July 20, 2017
Revised March 13, 2018
Accepted April 7, 2018

Keyword:
Network Function
Modeling
Network Functions
Virtualization
Performance Estimation

ABSTRACT

This work introduces a methodology for the modelization of network functions focused on the identification of recurring execution patterns as basic building blocks and aimed at providing a platform independent representation. By mapping each modeling building block on specific hardware, the performance of the network function can be estimated in terms of maximum throughput that the network function can achieve on the specific execution platform. The approach is such that once the basic modeling building blocks have been mapped, the estimate can be computed automatically for any modeled network function. Experimental results on several sample network functions show that although our approach cannot be very accurate without taking in consideration traffic characteristics, it is very valuable for those application where even loose estimates are key. One such example is orchestration in network functions virtualization (NFV) platforms, as well as in general virtualization platforms where virtual machine placement is based also on the performance of network services offered to them. Being able to automatically estimate the performance of a virtualized network function (VNF) on different execution hardware, enables optimal placement of VNFs themselves as well as the virtual hosts they serve, while efficiently utilizing available resources.

Copyright © 2018 Institute of Advanced Engineering and Science.
All rights reserved.

Corresponding Author:
Amedeo Sapio
Department of Control and Computer Engineering
Politecnico di Torino, Italy
amedeo.sapio@polito.it

1. INTRODUCTION

For a few years now software network appliances have been increasingly deployed. Initially, their appeal stemmed from their lower cost, shorter time-to-market, ease of upgrade when compared to purposely designed hardware devices. These features are particularly advantageous in the case of appliances, a.k.a. middleboxes, operating on relatively recent, higher layer protocols that are usually more complex and are possibly still evolving. More recently, with the overwhelming success and diffusion of cloud computing and virtualization, software appliances became natural means to ensure that network functionalities have the same flexibility and mobility as the virtual machines (VMs) they offer services to. In this context, implementing in software even less complex, more stable network functionalities is valuable. This trend led to embracing *Software Defined Networking* and *Network Functions Virtualization* (NFV). The former as a hybrid hardware/software approach to ensure high performance for lower layer packet forwarding, while retaining a high degree of flexibility and programmability. The latter as a virtualization solution targeting the execution of software network functions in isolated VMs sharing a pool of hosts, rather than on dedicated hardware (i.e., appliances). Such a solution enables virtual network appliances (i.e., VMs executing network functions) to be provisioned, allocated a different amount of resources, and possibly moved across data centers in little time, which is key in ensuring that the network can keep up with the flexibility in the provisioning and deployment of virtual hosts in today's virtualized data centers. Additional flexibility is offered when coupling NFV with SDN as network traffic can be steered through a chain of *Virtualized Network Functions* (VNFs) in order to provide aggregated services. With inputs from the industry, the NFV approach has been standardized by the European Telecommunications Standards Institute (ETSI) in 2013 [1].

The flexibility provided by NFV requires the ability to effectively assign compute nodes to VNFs and allocate the most appropriate amount of resources, such as CPU quota, RAM, virtual interfaces. In the ETSI standard the component in charge of taking such decisions is called *orchestrator* and it can also dynamically modify the

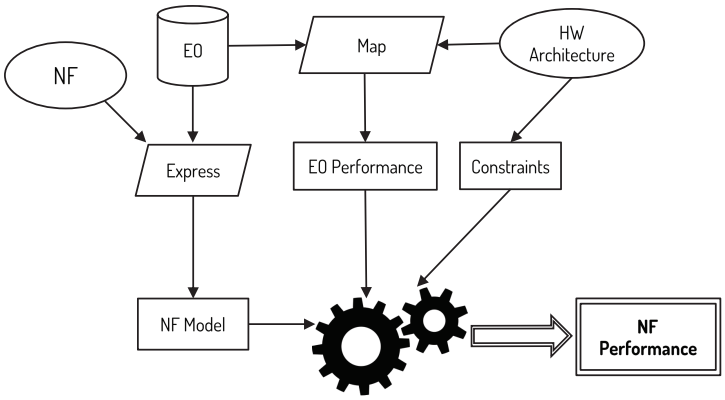

Figure 1. NF modeling and performance estimation approach.

amount of resources assigned to a running VNF when needed. The orchestrator can also request the migration of a VNF when the current compute node executing it is no longer capable of fulfilling the VNF performance requirements. These tasks require the orchestrator to be able to estimate the performance of VNFs according to the amount of resources they can use. Such estimation must take into account the nature of the traffic manipulation performed by the VNF at hand, some specifics of its implementation, and the expected amount of traffic it operates on. A good estimation is key in ensuring higher resource usage efficiency and avoid adjustments at runtime.

This work proposes a unified modeling approach applicable to any VNF, independently of the platform it is running on. By mapping a VNF model on a specific hardware it is possible to predict the maximum amount of traffic that the VNF can sustain with the required performance. The proposed modeling approach relies on the identification of the most significant operations performed by the VNF on the most common packets. These operations are described in a hardware independent notation to ensure that the model is valid for any execution platform. The mapping of the model on a target hardware architecture (required in order to determine the actual performance) can be automated, hence allowing to easily apply the approach to each available hardware platform and choose the most suitable for the execution.

Even if the proposed modeling approach has been defined with NFV in mind, it can be applied to non-virtualized network functions (NFs), whether implemented in software or hardware, provided that the implementation and characteristics of the underlying hardware are known. The availability of a unified modeling approach for VNF and NF is instrumental in the integration of middleboxes in an NFV infrastructure [2], which is important in a transition phase and for specific applications where a dedicated or specialized hardware platform is necessary for a specific NF to satisfy performance requirements.

The modeling approach is introduced in Section 2. and is illustrated in Section 3. by applying it to various network functions. In order to validate the proposed models, Section 4. compares the estimated performance with actual measurements of software network functions running on a general purpose hardware platform. After discussing related work in Section 5., Section 6. concludes the paper.

2. METHODOLOGY

The proposed modeling approach is based on the definition of a set of processing steps, here called *Elementary Operations* (EOs), that are common throughout various NF implementations. This stems from the observation that, generally, most NFs perform a rather small set of operations when processing the average packet, namely, well-defined alteration of packet header fields, coupled with data structure lookups.

An EO is informally defined as the longest sequence of elementary steps (e.g., CPU instructions or ASIC transactions) that is common among the processing tasks or multiple NFs. As a consequence, an EO has variable granularity ranging from a simple I/O or memory load operation, to a whole IP checksum computation. On the other hand, EOs are defined so that each can be potentially used in multiple NF models.

An NF is modeled as a sequence of EOs that represent the actions performed for the vast majority of packets. Since we are interested in performance estimation, we ignore operations that affects only a small number of packets (i.e., less the 1%), since these tasks have a negligible impact on performance, even when they are more complex and resource intensive than the most common ones. Accordingly exceptions, such as failures, configuration changes, etc., are not considered.

It is important to highlight that NF models produced with this approach are hardware independent, which ensures that they can be applied when NFs are deployed on different execution platforms. In order to estimate the

Table 1. List of sample EOs

	EO	Parameters	Description
1	I/O_mem - mem_I/O	hdr, data	Packet copy between I/O and (cache) memory
2	parse - deparse	b	Parse or encapsulate a data field
3	increase - decrease	b	Increase/decrease a field
4	sum	b	Sum 2 operands
5	checksum - inc_checksum	b	Compute IP checksum
6	array_access	es, max	Direct access to a byte array in memory
7	ht_lookup	N, HE, max, p	Simple hash table lookup
8	lpm_lookup	b, es	Longest prefix match lookup
9	ct_insertion	N, HE, max, p	Cache table insertion

performance of an NF on a specific hardware platform, each EO must be *mapped* on the hardware components involved in its execution and their features. This mapping allows to take into consideration the limits of the involved hardware components and gather a set of parameters that affect the performance (e.g., clock frequency). Moreover, the load incurred by each component when executing each EO must be estimated, whether through actual experiments or based on nominal hardware specifications. The data collected during such mapping are specific to EOs and the hardware platform, but not to a particular NF. Hence, they can be applied to estimate the performance of any NF modeled in terms of EOs. Specifically, the performance of each individual EO involved in the NF model is computed and composed considering the cumulative load that all EOs impose on the hardware components of the execution platform, while heeding all of the applicable constraints. Figure 1 summarizes the steps and intermediate outputs of the proposed approach.

Table 1 presents a list of sample EOs that we identified when modeling a number of NFs. Such list is by no means meant to be exhaustive; rather, it should be incrementally extended whenever it turns out that a new NF being considered cannot be described in terms of previously identified EOs. When defining an EO, it is important to identify the parameters related to traffic characteristics that significantly affect the execution and resource consumption.

2.1. Elementary Operations

A succinct description of the EOs listed in table 1 is provided below.

1. *Packet copy between I/O and memory*:

   A packet is copied from/to an I/O buffer to/from memory or CPU cache. *hdr* is the number of bytes that are preferably stored in the fastest cache memory, while *data* bytes can be kept in lower level cache or main memory. The parameters have been chosen taking into consideration that some NPUs provide a manual cache that can be explicitly loaded with the data that need fast access. General purpose CPUs may have assembler instructions (e.g., `PREFETCHh`) to explicitly influence the cache logic.

2. *Parse or encapsulate a data field*:

   A data field of *b* bytes stored in memory is parsed. A parsing operation is necessary before performing any computation on a field (it corresponds to loading a processor register). The dual operation, i.e., *deparse*, implies storing back into memory a properly constructed sequence of fields.

3. *Increase/decrease a field*:

   Increase/decrease the numerical value contained in a field of *b* bytes. The field to increase/decrease must have already been parsed.

4. *Sum 2 operands*:

   Two operands of *b* bytes are added.

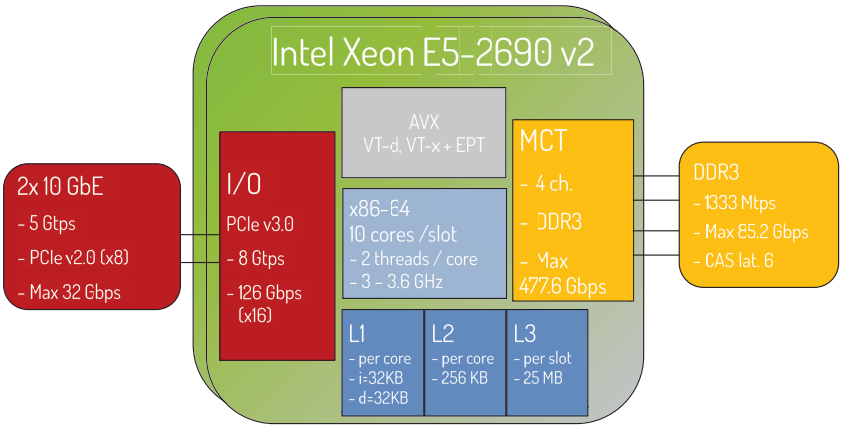

Figure 2. Hardware architecture description.

5. *Compute IP checksum*:

   The standard IP checksum computation is performed on *b* bytes. When only some bytes change in the relevant data, the checksum can be computed incrementally from the previous correct value [3]. In this case, the previous value of the checksum must be parsed beforehand and *b* is the number of changed bytes for which the checksum must be incrementally computed.

6. *Direct access to a byte array in memory*:

   This EO performs a direct access to an element of an array in memory using an index. Each array entry has size *es*, while the array has at most *max* entries.

7. *Simple hash table lookup*:

   A simple lookup in a direct hash table is performed. The hash key consists of *N* components and each entry has size equal to *HE*. The table has at most *max* entries and the collision probability is *p*.

8. *Longest Prefix Match lookup*:

   This EO selects an entry from a table based on the Longest Prefix Match (LPM). This lookup algorithm selects the most specific of the matching entries in a table (i.e., the one where the largest number of leading bits of the key match those in the table entry). The parameter *b* represents the number of bytes, on average, of the matching prefix, while *es* is the entry size.

9. *Cache table insertion*: Save in a hash table an entry with the current timestamp or update the timestamp if the entry is already present. This EO have the same parameters of the simple hash table lookup operation; the performance of both EOs depends from the hash table characteristics.

For the sake of simplicity (and without affecting the validity of the approach, as shown by the results in Section 4.), in modeling NFs by means of EOs, we assume that the number of processor registers is larger than the number of packet fields that must be processed simultaneously. Therefore there is no competition for processor registers.

2.2. Mapping to Hardware

We now proceed to map the described EOs to a specific hardware platform: a server with 2 Intel Xeon E5-2690 v2 CPUs (Ivy Bridge architecture with ten physical cores at 3 GHz), 64 GB DDR3 RAM memory and one Intel 82599ES network card with 2x10Gbps Ethernet ports. Figure 2 provides a schematic representation of the platform main components and relative constraints using the template proposed in [4].

Using the CPU reference manual [5], it is possible to determine the operations required for the execution of each EO in Table 1 and estimate the achievable performance.

1. `I/O_mem(hdr, data)-mem_I/O(hdr, data)`

The considered CPU provides a DMA-like mechanism to move data from the I/O buffers to the shared L3 cache and viceversa. Intel DPDK drivers [6] with Data Direct I/O Technology (DDIO) leverage this capability to move packets to/from the L3 cache without the CPU intervention, improving the packet processing speed. The

portion of each packet that must be processed (hdr) is then moved from L3 cache into the L1/L2 cache by the CPU. This operation requires 31 clock cycles to access the L3 cache, around 5 cycles to write a L1/L2 cache line and 9 cycles to write back a L3 cache line [7]. On the whole, the execution of this EO requires:

$$31 + [5|9] * \lceil \frac{hdr}{64B} \rceil \text{ clock cycles}$$

provided that hdr is less than the total amount of L1 and L2 caches, as it is reasonable for modern systems and common packet sizes. The multiplier is 5 for `I/O_mem` and 9 for `mem_I/O`.

2. `parse(b) - deparse(b)`

Loading a 64 bit register requires 5 clock cycles if data is in L1 cache or 12 clock cycles if data is in L2 cache, otherwise an additional L3 cache or DRAM memory access is required to retrieve a 64 byte line and store it in L1 or L2 respectively (the reverse operation has the same cost):

$$5 * \lceil \frac{b}{8B} \rceil \text{ clock cycles } \{+\lceil \frac{b}{64B} \rceil \text{ L3 or DRAM accesses}\}$$
$$\text{or}$$
$$12 * \lceil \frac{b}{8B} \rceil \text{ clock cycles } \{+\lceil \frac{b}{64B} \rceil \text{ L3 or DRAM accesses}\}$$

3. `increase(b) - decrease(b)`

Whether a processor includes an increase (decrease) instruction or one for adding (subtract) a constant value to a 64 bit register, this EO requires 1 clock cycle to complete. However, thanks to pipelining, up to 3 independent such instructions can be executed during 1 clock cycle:

$$\lceil 0.33 * \frac{b}{8B} \rceil \text{ clock cycles}$$

4. `sum(b)`

On the considered architecture, the execution of this EO is equivalent to the `increase(b)` EO. Please note that this is not necessarily the case on every architecture.

5. `checksum(b) - inc_checksum(b)`

Figure 3 shows a sample assembly code to compute a checksum on an Intel x86-64 processor. Assuming that the data on which the checksum is computed is not in L1/L2 cache, according to the Intel documentation [5], the execution of this code requires

$$7 * \lceil \frac{b}{2} \rceil + 8 \text{ clock cycles}$$
$$+\lceil \frac{b}{64B} \rceil \text{ L3 or DRAM accesses}$$

6. `array_access(es, max)`

Direct array access needs to execute an "`ADD`" instruction (1 clock cycle) for computing the index and a "`LOAD`" instruction resulting into a direct memory access and as many clock cycles as the number of CPU registers required to load the selected array element:

$$1 + \lceil \frac{es}{8B} \rceil \text{ clock cycles } + \lceil \frac{es}{64B} \rceil \text{ DRAM accesses}$$

7. `ht_lookup(N, HE, max, p)`

We assume that a simple hash table lookup is implemented according to the pseudo-code described in [4] and shown in Figure 4 for ease of reference.

Considering that the hash entry needs to be loaded from memory to L1 cache, a simple hash table lookup would require approximately:

$$\lceil (4 * N + 106 + 5 * \lceil \frac{HE}{8B} \rceil + 5 * \lceil \frac{HE}{32B} \rceil) * (1 + p) \rceil$$

```
Register ECX: number of bytes b
Register EDX: pointer to the buffer
Register EBX: checksum

CHECKSUM_LOOP:

    XOR EAX, EAX ;EAX=0
    MOV AX, WORD PTR [EDX] ;AX <- next word
    ADD EBX, EAX ;add to checksum
    SUB ECX, 2 ;update number of bytes
    ADD EDX, 2 ;update buffer
    CMP ECX, 1 ;check if ended
    JG CKSUM_LOOP

MOV EAX, EBX ;EAX=EBX=checksum
;EAX=checksum>>16 EAX is the carry
SHR EAX, 16
AND EBX, 0xffff ;EBX=checksum&0xffff
;EAX=(checksum>>16)+(checksum&0xffff)
ADD EAX, EBX
MOV EBX, EAX ;EBX=checksum
SHR EBX, 16 ;EBX=checksum>>16
ADD EAX, EBX ;checksum+=(checksum>>16)
MOV checksum, EAX ;checksum=EAX
```

Figure 3. Sample Intel x86 assembly code for checksum computation.

clock cycles and

$$\lceil(\lceil\frac{HE}{64B}\rceil * (1+p))\rceil \text{ L3 or DRAM accesses}$$

Otherwise, if the entry is already in the L1/L2 cache, the memory accesses and cache store operations are not required. Notice that in order for the whole table to be in cache, its size should be limited by:

$$max * HE \leq 32KB + 256KB = 288KB$$

8. `lpm_lookup(b,es)`

There are several different algorithms for finding the longest matching rule. Here we consider the *DIR-24-8* algorithm [8], which in most cases (when the entry matches up to 24 bits) is able to find the first matching rule with only one memory access. This speed, however, comes at the cost of space, because of the redundant storage of rules. However, the very fast lookup this algorithm provides heavily outweighs this space constraint. With the *DIR-24-8* algorithm the longest prefix match requires the equivalent of an `array_access(es,16M)` operation if $b \leq 3$, otherwise an additional memory access is required, corresponding to an `array_access(es,255)`.

9. `ct_insertion(N, HE, max, p)`

The EO corresponds to a lookup in a hash table followed by either the insertion of a new entry or the update of the timestamp in an existing one. The two operations have approximately the same cost; the pseudo-code in Figure 5 shows the operations required to update the timestamp of the entry. As a result the cache table insertion algorithm would require approximately:

$$\lceil(4 * N + 129 + 7 * \lceil\frac{HE}{8B}\rceil + 5 * \lceil\frac{HE}{32B}\rceil) * (1+p)\rceil$$

clock cycles and

$$2 * \lceil(\lceil\frac{HE}{64B}\rceil * (1+p))\rceil \text{ L3 or DRAM accesses}$$

3. MODELING USE CASES

This section demonstrates the application of the modeling approach described in section 2.. EOs are used to describe the operation of simple network functions, such as L2 Switches, and a more complex case, a *Broadband*

```
Register $1-N: key components
Register $HL: hash length
Register $HP: hash array pointer
Register $HE: hash entry size
Register $Z: result

Pseudo code:
    # hash key calculation
    eor $tmp, $tmp
    for i in 1 ... N
       eor $tmp, $i
    # key is available in $tmp

    # calculate hash index from key
    udiv $tmp2, $tmp, $HL
    mls $tmp2, $tmp2, $HL, $tmp
    # index is available in $tmp2

    # index -> hash entry pointer
    mul $tmp, $tmp2, $HE
    add $tmp, $HP
    # entry pointer available in $tmp

    <prefetch entry to L1 memory>
    # pointer to L1 entry -> $tmp2

    # hash key check (entry vs. key)
    for i in 1 ... N
       ldr $Z, [$tmp2], #4
       # check keys
       cmp $i, $Z
       bne collision
    # no jump means matching keys
    # pointer to data available in $Z
```

Figure 4. Hash table lookup pseudo-code.

Network Gateway (BNG). The model is used to estimate the performance of each use case on the hardware platform presented in Section 2.2.. The accuracy of the estimation is evaluated in Section 4. based on real measurements obtained through a range of experiments.

3.1. L2 Switch

First we model an Ethernet switch with a static forwarding table. In this case the output port is selected through a simple lookup in the table using the destination MAC address. Afterwards we consider a more general case where the forwarding table is populated using the backward learning algorithm. Finally, we model an MPLS switch, which selects the output interface according to the MPLS label in the packet.

3.1.1. Basic Forwarding

For each packet the switch selects the output interface where it must be forwarded; such interface is retrieved from a hash table using as a key the destination MAC address extracted from the packet.

More in detail, when a network interface receives a packet, it stores it in an I/O buffer. In order to access the Ethernet header, the CPU/NPU must first copy the packet in cache or main memory (possibly with the help of a Direct Memory Access module). Since the switch operates only on the Ethernet header together with the identifier of the ingress and egress ports through which it is received and forwarded, the corresponding 30 bytes ($18 + 6 + 6$ bytes)[1] are copied in the fastest cache, while the rest of the packet (up to 1500 bytes) can be kept in L3 cache or

[1] In this paper we consider that interfaces are identified by their Ethernet address. Different implementations can use a different identifier, which leads to a minor variation in the model.

```
 Register $HE: updated hash entry
 Register $HT: pointer to previous L1 entry
 Register $HS: hash entry size

Pseudo code:
    for i in 1 ... $HS/8
       mov [$HT], $HE
       add $HT, #8

    #update timestamp
    rdtsc
    mov [$HT], EDX
    add $HT, #2
    mov [$HT], EAX

    <store updated entry>
```

Figure 5. Entry update pseudo-code for cache table insertion.

I/O_mem(30,ps)
parse(6)
ht_lookup(1,12,2M,0)
deparse(6)
mem_I/O(30,ps)

(a) Basic forwarding switch model.

I/O_mem(30,ps)
parse(8)
ht_lookup(1,14,2M,0)
parse(12)
ct_insertion(2,14,2M,0)
deparse(6)
mem_I/O(30,ps)

(b) Learning switch model.

I/O_mem(34,ps-4)
parse(3)
ht_lookup(1,12,1M,0)
parse(1)
decrease(1)
deparse(10)
mem_I/O(34,ps-4)

(c) MPLS switch model.

Figure 6. Models of different L2 switches.

main memory. To ensure generality, we consider that an incoming packet cannot be copied directly from an I/O buffer to another, but instead it must be first copied in (cache) memory.

The switch must then read the destination MAC address (6 bytes) prior to using it to access the hash table to get the appropriate output interface. The hash table has one key (the destination MAC) and consists of 12 byte entries composed of the key and the output interface MAC address. A common number of entries in a typical switch implementation is $\approx 2M$, which gives an idea, when mapping the model to a specific hardware, of whether the hash table can be fully stored in cache under generic traffic conditions. The new output port must be stored in the data structure in L3 cache or main memory (which, as previously explained, has the same cost as parsing 6 bytes), before moving the packet to the buffer of the selected output I/O device.

The resulting model expressing the above steps in terms of EOs is summarized in Figure 6a, where `ps` is the ethernet payload size. Such model assumes that the collision probability of the hash is negligible (i.e., the hash table is sufficiently sparse).

Applying to the Ethernet switch model the mapping of EOs presented in Section 2.2., we can estimate that forwarding a packet, regardless of the packet size (thanks to DDIO), requires:

$$213 \text{ clock cycles } + 1 \text{ DRAM access}$$

As a consequence, a single core of an Intel Xeon E5-2690v2 operating at *3.6 Ghz* can process $\approx$ *17.31 Mpps*, while the DDR3 memory can support *111.08 Mpps*. The memory throughput is estimated considering that each packet requires a 12 byte memory access to read the hash table entry, which has a latency of:

$$\frac{(\text{CAS latency} \times 2) + 3}{\text{data rate}}$$

If we consider minimum size (64 bytes) packets (i.e., an unrealistic, worst case scenario), a single core can process $\approx$ *11.36 Gbps*.

3.1.2. Learning Switch

We here consider an Ethernet switch with VLAN support, in which case the key used for lookups in the forwarding table consists of the destination MAC address and the VLAN ID (2 bytes). Hence, 8 bytes must be parsed from the header (destination address and VLAN ID) of each packet in order to obtain the lookup key and entries in the forwarding table are 14 bytes long (destination address and VLAN ID as key and output interface as value). Since the switch is applying backward learning, for each packet the source MAC address and source port are used to update the forwarding table. The switch must also parse the source MAC address and read from memory the source port (added to packets stored in memory) and either add an entry in the forwarding table or just update the timestamp of an existing one. The resulting model is shown in Figure 6b.

When mapped to our hardware architecture, forwarding a packet requires an estimated:

$$352 \text{ clock cycles } + 2 \text{ DRAM accesses}$$

hence the maximum throughput reachable by a single core is reduced to $\approx$ *10.47 Mpps*, while the DDR3 memory can support *55.54 Mpps*. This translates to a maximum throughput of $\approx$ *6.87 Gbps* for 64 byte packets.

3.1.3. MPLS Switch

An MPLS switch is a simple, yet currently widely deployed, Network Function. For each packet the switch swaps a single MPLS label and forwards the packet on an Ethernet network towards the next hop. The new label and the next hop are retrieved from a hash table whose key is the label extracted from the packet. Since the MPLS switch modifies the label in the MPLS header, in addition to associating to it the output port, the MPLS header (4 bytes) is also preferably copied in the L1/L2 cache, while the rest of the packet can be kept in L3 cache or main memory. The switch must then extract the MPLS label (20 bit $\approx$ 3 bytes) prior to using it to access the hash table to get the new label and the next hop. The hash table has one key (the label) and consists of 12 byte entries:

- Input label (key) - 3 bytes
- Output label - 3 bytes
- Next hop Ethernet address - 6 bytes.

The maximum number of entries in the hash table is, in the worst case, $1M = 2^{20}$ and we consider that the collision probability is negligible.

In the most general case, each entry, referred in the MPLS standard documents as Next Hop Label Forwarding Entry (NHLFE), could hold more than one label in case of multiple label operations. For the sake of simplicity we model only a single label operation: the swapping of a label, which is the most frequent operation in common MPLS switch deployment scenarios.

The switch must also decrease the Time-To-Live (TTL) contained in the MPLS header, which requires parsing the corresponding field, followed by a decrease operation for the 1 byte field. The new (outgoing) MPLS header and output port must be stored in main memory (encapsulation of 10 bytes) and moved to the buffer of the output I/O device. The resulting model is summarized in Figure 6c.

As we map this model to the considered hardware platform, we can conclude that the estimated forwarding cost for a MPLS switch is:

$$224 \text{ clock cycles } + 1 \text{ DRAM access}$$

corresponding to a maximum per core throughput of $\approx$ *16.45 Mpps*, while the memory could provide the same throughput as the basic forwarding switch. The maximum bitrate considering 64 bytes packets is $\approx$ *10.8 Gbps*.

3.2. Broadband Network Gateway

A Broadband Network Gateway (BNG) is the first IP point in the network for DSL and cable modem subscribers connecting them to the broadband IP network. The primary task of a BNG is to aggregate traffic from various subscriber sessions from an access network, and route it to the core network of the service provider. Moreover, a BNG carries out additional vital tasks for Network Service Providers (NSPs), such as managing subscribers' sessions, performing accounting and enforcing operator policies. Hence, a BNG represents a more complex use case for the application of the proposed modelization approach.

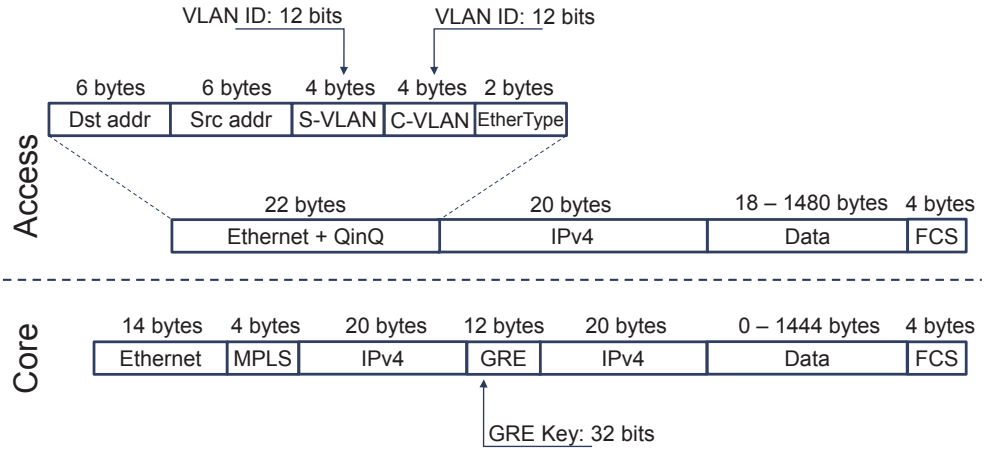

Figure 7. Packet formats.

In our modeling effort we refer to the software implementation of a BNG present in the Intel Data Plane Performance Demonstrators (DPPD) [9]. This is an open source, highly optimized software BNG specifically intended for performance analysis. In this implementation the traffic in the access network between the Customer Premise Equipment (CPE) and the BNG is encapsulated using Ethernet QinQ frames, while the traffic between the BNG and the Carrier-grade NAT (CGNAT) in the core MPLS network is encapsulated using GRE (Generic Routing Encapsulation). In this scenario packets received from the access network and packets received from the core network are processed differently by the BNG, thus 2 separate models are required for the 2 directions. The two different formats of packets forwarded in the access and in the core network is illustrated in Figure 7.

Packets from CPEs are matched with 2 different tables: (*i*) a hash table that given the QinQ tag provides the corresponding GRE key (up to 16M entries of 7 bytes) and (*ii*) an LPM routing table that given the destination IP address returns the output port, the IP address of the remote GRE tunnel endpoint, the next hop MAC address and the MPLS label (this table can contain up to 8K routes). Packets from the core network are instead matched with only one hash table that given the GRE key and the inner destination IP address provides the QinQ tag, the destination MAC address and the output port. The BNG supports up to 64K CPEs, thus this table can contain up to 64K entries of 23 bytes. The QinQ tag and the GRE key are used to track the subscriber (e.g., for accounting), while the tunnel endpoint (i.e., the CGNAT) is selected according to the destination of the packet.

The resulting models for both directions are summarized in Figure 8. When processing packets from the access network, MAC with QinQ and IP headers are loaded preferably in L1/L2 cache, so that the QinQ header can be parsed. The extracted QinQ tag is used for the lookup in table (*i*), while the destination IP address is parsed and deployed in the LPM lookup table (*ii*). These 2 lookups provide the output GRE key, destination IP and MAC addresses, MPLS tag and output port that are used in the encapsulation of the output packet. The TTL (Time To Live) of the internal IP packet is decremented and thus the checksum must be incrementally updated starting from the current value. The new packet format requires also the computation of the GRE checksum and the external IP packet Total Length field and header checksum. Moreover, backward learning is used to populate the table used to process packets from the core network. Hence, an additional `ct_insertion` operation is required, after parsing source port, MAC and IP addresses. The final packet is formed with the encapsulation of 70 bytes, corresponding to the new ethernet, MPLS, external IP, GRE and inner IP headers and then sent to the output I/O buffer.

Packets from the core network require a parse operation for the GRE key and the inner destination IP before using them for an hash table lookup to get the QinQ tag, the destination MAC address and the output port. In this case also the TTL of the inner IP packet is decremented and the checksum incrementally updated. The new outgoing packet must then be stored in memory or cache (encapsulation of 42 bytes) and moved to the buffer of the output I/O device.

Mapping these models to the considered hardware platform, we can conclude that the estimated cost to process a 64 bytes packet from the access network is:

$$717 \text{ clock cycles } + 6 \text{ DRAM accesses}$$

corresponding to a maximum per core throughput of $\approx$ *5.14 Mpps* (*3.37 Gbps*), while the DDR3 memory can support $\approx$ *12.11 Mpps* (*7.95 Gbps*). The estimated cost to process a 64 byte packet from the core network is:

$$274 \text{ clock cycles } + 1 \text{ DRAM access}$$

Packet from access network
I/O_mem(42,ps-20)
parse(8)
ht_lookup(1,7,16M,0)
parse(4)
lpm_lookup(2,23)
parse(1)
decrease(1)
parse(2)
inc_checksum(1)
checksum(ps-14)
sum(2)
checksum(20)
parse(16)
ct_insertion(2,23,64K,0)
deparse(70)
mem_I/O(70,ps-20)

Packet from core network
I/O_mem(70,ps-56)
parse(8)
ht_lookup(2,23,64K,0)
parse(1)
decrease(1)
parse(2)
inc_checksum(1)
deparse(42)
mem_I/O(42,ps-56)

Figure 8. BNG model.

corresponding to a maximum per core throughput $\approx$ *13.45 Mpps* (*8.83 Gbps*) and $\approx$ *24.68 Mpps* (*16.2 Gbps*) achievable by the DDR3 memory.

4. EXPERIMENTAL VALIDATION

In order to evaluate the accuracy of the estimates produced by the proposed modeling approach, in this section we present measurements made in a lab setting with software implementations of the presented Network Functions.

4.1. L2 Switch

As a software L2 switch we deploy an instance of Open vSwitch [10] configured through the OpenFlow protocol to select an output port based on the destination MAC address. The switch is used with both a predefined forwarding table and backward learning. Moreover, the same switch implementation is also configured to perform MPLS label swapping. The software switch runs on the hardware platform presented in Figure 2.

To minimize the interference of the operating system drivers, the network interfaces are managed through the Intel DPDK drivers [6]. These drivers are designed for fast packet processing, providing the possibility to receive and send packets directly from/to a network interface card within the minimum possible number of CPU cycles. In fact, DPDK drivers allow the CPU to receive packets using polling, rather than interrupts, since interrupt service routines execute a number of additional operations for each packet. Moreover, with DPDK drivers it is possible to leverage DDIO to load packets directly in the L3 cache with no overhead for the CPU.

A separate PC with the same hardware configuration is used as a traffic generator leveraging PF_RING/DNA drivers [11] to generate traffic up to the link capacity even with packets of minimum size. As shown in Figure 9, we run 4 different processes, 2 PF_RING senders and 2 PF_RING counters, pinned on different dedicated cores, to generate traffic on both NICs at line rate and, at the same time, compute statistics on received packets. All the tests are run for 5 minutes and the results present the averaged aggregate statistics on both sinks.

4.1.1. Basic Forwarding

To test the forwarding performance of the software switch, we generate traffic consisting of Ethernet packets with ever different destination MAC addresses, in order to prevent inter-packet caching. For each destination address we had previously added a rule in the switch to set the destination port. The resulting throughput for different packet sizes is presented in Figure 10, together with the values estimated with the modeling approach in Section 3.1.1..

The experimental results show that in this scenario the switch can achieve throughput up to the link capacity except with packets smaller than 128 bytes. For values of the performance estimate that exceed the link capacity (i.e.,

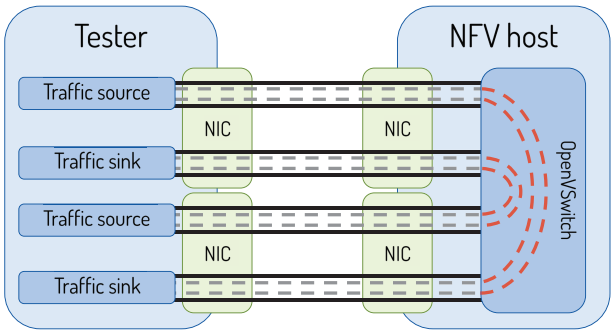

Figure 9. L2 Switch testbed setup.

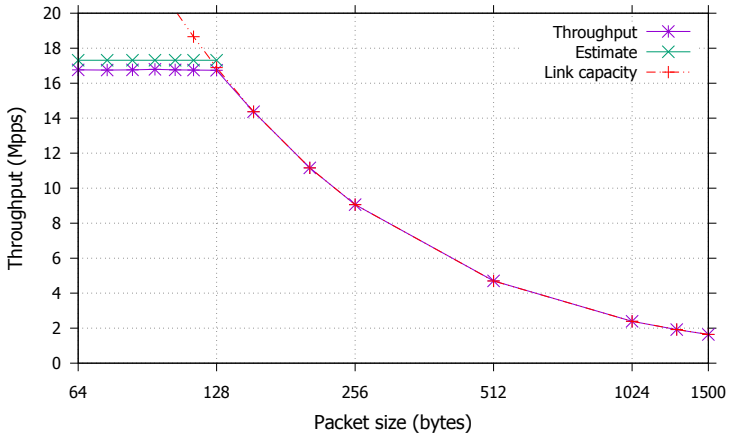

Figure 10. Basic forwarding performance.

packets greater than 128 bytes), our model cannot be applied by itself as it considers the hardware computational capability and not the transmission rate of the physical links that becomes the limiting factor in such scenario. With smaller packets, our mode estimates a rate around *17 Mpps*, regardless of the actual packet size. The measurements demonstrate that that the throughput estimation is quite accurate, with only a 4% error.

4.1.2. Learning Switch

The next test is aimed at measuring to what extent the performance of the software switch is impacted in a context in which the learning algorithm plays a significant role in the processing being performed. We configure the switch by pushing an OpenFlow rule with a "`NORMAL`" action, so that it acts as a regular layer 2 learning switch [12]. Then, before starting the test, for each destination address that will be used in the test traffic, the corresponding traffic sink sends a packet with the same address as source. This allows the switch to learn the output port associated with the addresses to ensure that measurements will be taken in a steady state (i.e., avoiding that some packets are flooded on all ports, while others are sent out on a specific port).

To isolate the impact of caching on the performance we consider 2 scenarios. In a first test each traffic source sends traffic addressed to only one destination and with a unique source address. In a second test each traffic source sends traffic using repeatedly 10 different source and destination addresses. We chose this number of different address pairs after a preliminary evaluation, which showed that this is the turning point at which the throughput experiences a sharp decrease due to cache misses. The difference between the basic forwarding throughput (see Figure 10) and the throughput in this second scenario represents the performance degradation due to the learning functionality itself. The resulting throughput measured in both tests by the traffic sinks, is presented in Figure 11 for different packet sizes, together with the values estimated with the proposed model.

The results show that, when all the packets have a single source and destination address pair, the switch can achieve a very high throughput ($\approx$ *16 Mpps* with packets that are 128 bytes or smaller) because the 2 corresponding entries (one for forwarding and one for learning) are matched within the *microflow* cache [10] that Open vSwitch implements in kernel space. Since the microflow cache is stored in L1/L2 cache (thanks to its small size), the execution of the learning code updating the timestamp requires very few clock cycles ($\approx$ *140*), significantly lower

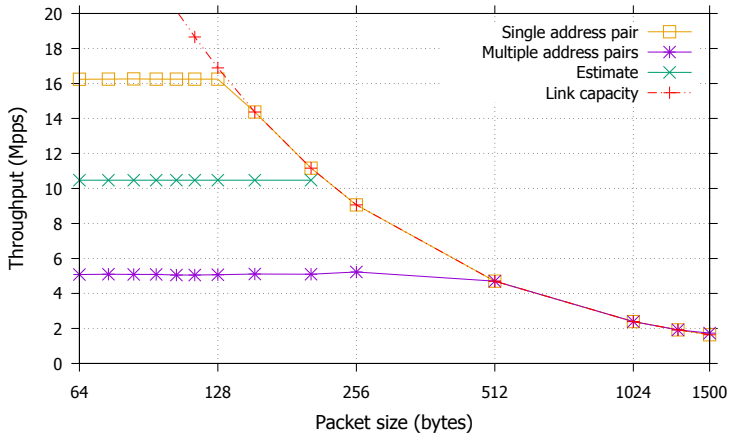

Figure 11. Learning switch performance.

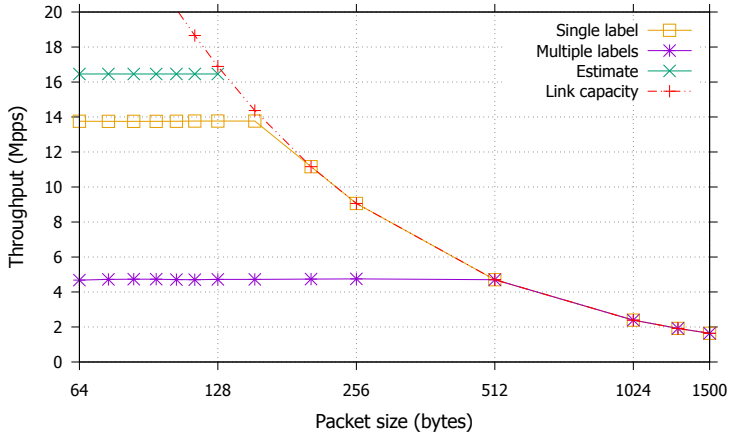

Figure 12. MPLS switch performance.

than our estimate ($\approx$ *350*) that assumes a main memory-based lookup of the forwarding table (because the Open vSwitch implementation specific microflow cache is not modeled).

On the contrary, with 10 different addresses the throughput radically drops to only *5 Mpps* for packets smaller than 512 bytes because the forwarding entry to be updated is not in the microflow cache, in which case the Open vSwitch implementation delegates the update operation to a user space process. This is significantly different from the estimated throughput of $\approx$ *10.5 Mpps* expected when an entry timestamp is updated for each packet. While the design choice of Open vSwitch performing packet processing, beyond basic forwarding, in user space increases flexibility and configurability, it adds a large overhead. As the test show, this complexity is not considered in the model. On the other hand, our model also does not capture the microflow cache-based optimization and the unlikely case in which it allows to avoid a lookup within the complete hash table. In fact, the modelization approach we are proposing aims at evaluating the operation of an optimized NF operating in average conditions.

4.1.3. MPLS Switch

We evaluate the MPLS Switch model presented in Section 3.1.3. using MPLS over Ethernet frames. As in the previous case, we perform 2 different tests, one where each source sends traffic with only one MPLS label, and a second test where each source sends packets marked with 10 different MPLS labels. A rule matching each label present in the source traffic is added in the switch before the test begins.

The results of the tests and the estimate plotted in Figure 12 show that in both scenarios the measured throughput is below the estimated value. Since MPLS packet processing is computationally very similar to basic forwarding, the model estimates a *16.76 Mpps* throughput that is close to the *17.47 Mpps* estimate for basic forwarding. On the contrary, the measured throughput for the MPLS switch when operating with multiple labels (in a scenario comparable to the basic forwarding tests) is well below (about one third) the one obtained with basic

forwarding, which hints to a poor optimization of MPLS processing in the software switch implementation. This might be related to the fact that forwarding based on MPLS labels was added to Open vSwitch relatively recently, hence the code is not as mature and optimized as the Ethernet address-based forwarding one. The large difference between the single and multiple labels tests shows that caching is playing an important role and in a real scenario, with traffic with multiple different labels, the software switch performance takes a significant hit (being almost one third of the case that takes advantage of caching).

4.2. Broadband Network Gateway

We run our tests on the BNG platform provided by Intel DPPD [9] on the hardware platform presented in Figure 2. This platform has been upgraded with one additional Intel 82599ES network card with 2x10Gbps Ethernet ports, given that the software requires 2 ports connected to the access network and 2 ports connected to the core network. The Intel Packet pROcessing eXecution Engine (PROX) is run on a second machine with the same hardware characteristics as a traffic generator. The test is run using the Intel Dataplane Automated Testing System (DATS), which controls one instance of PROX running on the tester machine to generate and to analyze the traffic and one instance of the BNG on the other host. DATS generates a realistic workload simulating traffic from 32K users per port and with 8K possible routes. The test is executed 10 times and the averaged results are presented in Figure 13, where they are compared with the estimate devised in Section 3.2.. Since DATS reports the aggregated throughput, corresponding to the total number of packets per second processed by the BNG, the plotted estimate is the average of the throughput estimates in the 2 directions.

The BNG software spawns 4 load balancer threads (one per interface) distributing the traffic among 6 packet processing threads. Therefore in the estimate we consider that 6 cores are dedicated to packet processing. Thanks to the parallel execution the CPU could theoretically process up to *42.5 Mpps* with 78 byte packets (the smallest packet generated by PROX). Hovewer, given the considerable number of main memory accesses required by the BNG to process a single packet, our model concludes that the overall throughput (for small packets) is limited by the memory latency and only *23.74 Mpps* can be processed when packets are 78 byte long. As shown by the `generic estimate` line plotted in Figure 13, the model is quite accurate in estimating the performance for small packets, with a 7% average error for packets up to 204 bytes. However, the model is less accurate for larger packet sizes. This is due to a side effect of the distributed execution of the NF.

The hardware platform used in the experiments has 2 processors and 2 NICs, each NIC connected to the socket of one of the processors. The DDIO mechanism considered in Section 2.2. when mapping the `I/O_mem(hdr, data)` EO onto the hardware platform, moves each packet received through a NIC to the L3 cache of the processor it is connected to. When a packet processing thread running on the other processor executes the `I/O_mem(hdr, data)` EO on a packet stored in the L3 cache of the other processor, its cost is different than the one presented in Section 2.2. because the processor must read the packet from the main memory and load it into its own L1/L2 cache before starting processing it. As a result, in this case the execution of the EO requires:

$$30 + 5 * \lceil \frac{hdr + data}{64B} \rceil \text{ clock cycles} + \lceil \frac{hdr + data}{64B} \rceil \text{ DRAM accesses}$$

Note that the BNG needs to process the whole packet (i.e., `hdr+data` bytes), not just the header, in order to compute the GRE checksum.

Considering that that the 4 load balancer threads are uniformly distributing packets on the 6 packet processing threads and that the traffic load on the two interfaces is the same, there is a 50% chance for a packet not to be in the L3 cache of the processor running the corresponding processing thread, which is taken into account in plotting the `platform specific estimate` line in Figure 13. When compared to the `generic estimate`, it provides a more accurate throughput estimate for larger packets that cause a non-negligible wait time for the processor retrieving them from main memory.

4.3. Concluding Remarks

The comparison of experimental results with the estimates produced by our model presented in this section shows that software NF performance, and consequently the modeling accuracy, are heavily affected by multiple quite specific factors, such as:

- The effectiveness of caching mechanisms realized in both the execution platform (e.g., processor cache) and the software implementation of algorithms and data structures (e.g., the microflow cache). Cache deployment largely increases the performance variability, which cannot be captured by a model since per-packet cost strongly depends on the traffic runtime characteristics.

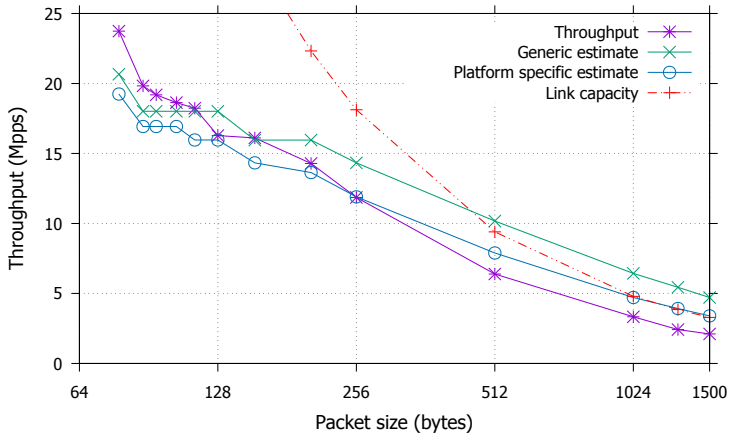

Figure 13. Broadband Network Gateway performance.

- The implementation of the NF. Our performance estimation approach is well suited to NFs designed to perform a specific, well-defined, packet processing operation at high speed. In this context, the Intel BNG proved to be a valid use case for which the model provides a good estimation, being able to identify the per-packet processing cost and the aspects limiting the maximum throughput. On the contrary, general purpose implementations based on a generic, configurable pipeline to process packets in multiple ways are not well modeled by our approach, as shown by the tests on Open vSwitch. To provide programmability and flexibility, general purpose implementations might perform for each packet a number of operations not specifically needed by the required function. Our experiments showed that only when Open vSwitch is configured to perform the simplest supported operation (i.e., basic forwarding of Ethernet frames), the performance is more predictable and correctly modeled with our approach.

- Parallel execution of operations. Our approach is not meant to model the interactions and dependencies among components running in parallel on different processors. As shown by the case of the BNG running 6 parallel threads on 2 processors, the model had to be specialized to take into account the specific scenario.

In summary, our experimental evaluation demonstrates that a generic model cannot fully capture all the aspects that can affect the performance of an NF, which could be achieved only by delving into a level of detail that would make the model extremely detailed and specific of a given implementation and instantiation for execution on a specific hardware platform. Hence, the model cannot be expected to estimate the performance of an NF with high accuracy. Such relatively loose estimate is anyway not worthless and has at least two very valuable applications: (*i*) in support of VM scheduling and VNF orchestration in cloud environments and (*ii*) as a reference performance upper bound in both the design and improvement of a NF software implementation.

Moreover, the proposed NF model can also be mapped on hardware implementations, in which case we expect the performance to have less variability and consequently the model to provide a more accurate estimate.

5. RELATED WORK

In this paper we present an updated version of the methodology first introduced in [13]. By applying the methodology to more complex use cases, as showed in Section 3., we have gained experience, which has led to improvements in the methodology itself. This paper offers also a more extensive experimental evaluation based on additional use cases.

This work was initially inspired by [4] that aims to demonstrate that the Software Defined Networks approach does not necessarily imply lower performance compared to purpose-built ASICs. In order to prove it, the performance of a software implementation of an Ethernet Provider Backbone Edge Bridge is evaluated. The execution platform considered in [4] is a hypothetical network processor, for which a high-level model is provided. Unlike our work, the authors do not aim at providing a universal modelization approach for generic network functions. Rather, their purpose is to leverage the usecase of a specific sample network function to demonstrate that, even for very specific tasks, the NPU-based software implementation offers performance only slightly lower than purpose designed chips.

[14] presents a modeling approach for describing packet processing in middleboxes and the ways they can be deployed. The approach is applied to a NAT, an L4 load balancer, and an L7 load balancer. The proposed model

is inherently different from ours in that it is not aimed at estimating performance and resource requirements, but it rather focuses on accurately describing functionalities to support decisions in the middlebox deployment.

Cloud platform management solutions that take into account the performance of the network infrastructure when placing VMs [15, 16, 17] could greatly benefit from a VNF performance estimate. For example, [17] describes the changes needed in the OpenStack software platform, the open-source reference cloud management system, to enable the Nova scheduler to plan VM allocation based on network properties and a set of constraints provided by the orchestrator. We argue that in order to enforce such constraints, the orchestrator needs a VNF model like the ones generated by the approach presented in our paper. However, the presented methodology cannot be applied as such to VNFs because the additional overhead introduced by virtualization must be considered. A few works addressed this specific aspect. [18] presents a generic model to predict performance overheads on various virtualization platforms, based on the evaluation of the most influencing factors, such as CPU scheduling and resource overcommitment, while in [19] the virtualization overhead is estimated with focus on the impact of sole resource contention. Resource usage of virtualized applications is addressed in [20] by means of regression models, starting from benchmark results. While these studies offer ways of estimating virtualized application performance, when considering an NFV environment it is essential to take into account the overhead related to the virtual switch in the hypervisor, which uses a relevant share of processor time to forward traffic to and from VNFs. Our modelization approach can be applied to devise an estimate of the resources required by the virtualized network function and inter-VMs traffic steering, thus enabling a more accurate VNF performance estimate.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a unified modeling approach aimed at performance estimation of Network Functions when executed on different platforms. Starting from the identification of the most relevant operations performed by the NF on the majority of packets, the presented methodology allows to define a platform independent model of such a NF. The model can then be automatically mapped to the target execution platform, leveraging the characterization of hardware performance. This methodology is especially helpful in planning VNFs placement and resources allocation, and is valuable for integration of middleboxes in an NFV infrastructure.

The presented experiment results show that the proposed modeling approach provides a way to obtain a usable, even though loose, estimate of NF performance, especially for single-purpose, highly optimized, software implementations. The results show also that a very accurate estimation cannot be obtained without taking in consideration characteristics of the traffic. We claim that the proposed modelization approach can be valuable for those application where the traffic profile is not known a priori, such as VNF scheduling and orchestration. Moreover, the model can be fine-tuned at runtime with the support of traffic and performance monitoring to adapt to the traffic profile. We plan, as future work, to integrate the modelization methodology with online refinement, leveraging live performance monitoring. We also plan to investigate the application of the modeling approach to estimate the performance of hardware NFs and to evaluate the performance cost of traffic steering in a cloud computing enviroment.

ACKNOWLEDGEMENT

This work was conducted within the framework of the FP7 UNIFY project, which is partially funded by the Commission of the European Union. Study sponsors had no role in writing this report. The views expressed do not necessarily represent the views of the authors' employers, the UNIFY project, or the Commission of the European Union. The authors wish to thank David Verbeiren and Intel for providing the testbed, Gergely Pongrácz and Ferenc Juhász for their valuable insights.

REFERENCES

[1] "ETSI ISG for NFV, ETSI GS NFV-INF 001, Network Functions Virtualisation (NFV); Infrastructure Overview," http://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/001/01.01.01_60/gs_NFV-INF001v010101p.pdf, [Online; accessed 18-February-2017].

[2] M. Baldi, R. Bonafiglia, F. Risso, and A. Sapio, "Modeling native software components as virtual network functions," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 605–606.

[3] A. Rijsinghani, "RFC 1624 - Computation of the Internet Checksum via Incremental Update," 1994. [Online]. Available: https://tools.ietf.org/html/rfc1624

[4] G. Pongrácz, L. Molnár, Z. L. Kis, and Z. Turányi, "Cheap silicon: a myth or reality? picking the right data plane hardware for software defined networking," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 103–108.

[1] http://www.fp7-unify.eu/

[5] "Intel 64 and IA-32 Architectures Optimization Reference Manual," http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html, [Online; accessed 18-February-2017].
[6] "Intel DPDK: Data Plane Development Kit," http://dpdk.org, [Online; accessed 19-March-2017].
[7] "Intel Ivy Bridge Benchmark," http://www.7-cpu.com/cpu/IvyBridge.html, [Online; accessed 18-February-2017].
[8] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 1998, pp. 1240–1247.
[9] "Intel Data Plane Performance Demonstrators," https://01.org/intel-data-plane-performance-demonstrators, [Online; accessed 18-February-2017].
[10] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch." in *NSDI*, 2015, pp. 117–130.
[11] "Ntop PF_RING," http://www.ntop.org/products/packet-capture/pf_ring/, [Online; accessed 19-March-2017].
[12] "Open vSwitch Manual - ovs-ofctl," http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt, [Online; accessed 19-March-2017].
[13] A. Sapio, M. Baldi, and G. Pongracz, "Cross-platform estimation of network function performance," in *Software Defined Networks (EWSDN), 2015 Fourth European Workshop on*. IEEE, 2015, pp. 73–78.
[14] D. Joseph and I. Stoica, "Modeling middleboxes," *Network, IEEE*, vol. 22, no. 5, pp. 20–25, 2008.
[15] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, "Stratos: A network-aware orchestration layer for middleboxes in the cloud," Technical Report, Tech. Rep., 2013.
[16] J. Soares, M. Dias, J. Carapinha, B. Parreira, and S. Sargento, "Cloud4nfv: A platform for virtual network functions," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*. IEEE, 2014, pp. 288–293.
[17] F. Lucrezia, G. Marchetto, F. G. O. Risso, and V. Vercellone, "Introducing network-aware scheduling capabilities in openstack," *Network Softwarization (NetSoft), 2015 IEEE 1st Conference on*, 2015.
[18] N. Huber, M. von Quast, M. Hauck, and S. Kounev, "Evaluating and modeling virtualization performance overhead for cloud environments." in *CLOSER*, 2011, pp. 563–573.
[19] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell, "Vm 3: Measuring, modeling and managing vm shared resources," *Computer Networks*, vol. 53, no. 17, pp. 2873–2887, 2009.
[20] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy, "Profiling and modeling resource usage of virtualized applications," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., 2008, pp. 366–387.

BIOGRAPHIES OF AUTHORS

Mario Baldi is Director of Technology at Cisco Systems and Associate Professor at Politecnico di Torino. He was Data Scientist Director at Symantec Corp., Inc., Principal Member of Technical Staff at Narus, Inc., Principal Architect at Embrane, Inc.; Vice Dean of the PoliTong Sino-Italian Campus at Tongji University, Shanghai; Vice President for Protocol Architecture at Synchrodyne Networks, Inc., New York. Through his research, teaching and professional activities, Mario Baldi has built considerable knowledge and expertise in big data analytics, next generation network data analysis, internetworking, high performance switching, optical networking, quality of service, multimedia networking, trust in distributed software execution, and computer networks in general.

Amedeo Sapio is a PhD student in Control and Computer Engineering within the networking research group (NetGroup) at Politecnico di Torino. He received the M.Sc. degree cum laude in Computer Engineering in 2014. His main research interests are high-speed packet processing, software defined networking and innovative network services.