An accurate and efficient modeling framework for the performance evaluation of DPDK-based virtual switches

Thomas Begin*, Bruno Baynat†, Guillaume Artero Gallardo‡ and Vincent Jardin§
*Univ Lyon, Université Claude Bernard Lyon 1, ENS de Lyon, Inria, CNRS, LIP, France
†Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6, France
‡ Sysoco Group, Research and Development, France
§ 6WIND, Research and Development, France

***Abstract*—DPDK works as a specialized library that enables virtual switches to accelerate the processing of incoming packets by, among other things, balancing the incoming flow of packets over all the CPU cores and processing packets by batches to make a better use of the CPU cache. Although DPDK has become a de facto standard, the performance modeling of a DPDK-based vSwitch remains a challenging problem.**

In this paper, we present an analytical queueing model to evaluate the performance of a DPDK-based vSwitch. Such a virtual equipment is represented by a complex polling system in which packets are processed by batches, i.e., a given CPU core processes several packets of one of its attached input queues before switching to the next one. To reduce the complexity of the associated model, we develop a general framework that consists in decoupling the polling system into several queueing subsystems, each one corresponding to a given CPU core. We resort to servers with vacation to capture the interactions between subsystems. Our proposed solution is conceptually simple, easy to implement and computationally efficient.

Tens of comparisons against a discrete-event simulator show that our models typically deliver accurate estimates of the performance parameters of interest (e.g., attained throughput, packet latency or loss rate). We illustrate how our models can help in determining an adequate setting of the vSwitch parameters using several real-life case studies.

***Index Terms*—NFV; virtual switch; DPDK; modeling; performance evaluation; polling system; batch.**

I. Introduction

Server virtualization has become ubiquitous in the modern IT environment. Decoupling virtual servers from physical servers helps to leverage the computing resources, and brings important gains in scalability and agility. More recently, the virtualization of networks has attracted much attention. Scalability, agility, and multi-tenancy (with the concept of network slicing) are the envisioned improvements that virtualization will bring to traditional computer networks.

This trend towards more flexible networks, often known as "softwarization", is driven by two main paradigms: Software-Defined Networking (SDN) and Network Function Virtualization (NFV). The former aims at removing all the decision-making networking functions from network nodes and re-grouping them into a (set of) controller(s). Thus, network nodes, such as routers, switches, load-balancers, firewalls, etc. are replaced by appliances receiving their instructions directly from the controller(s) (using a standard interface like OpenFlow [1]). On the other hand, NFV refers to the gradual move of network functions from dedicated hardware to commodity hardware running specialized software. For example, functions such as routing, switching, load-balancing, firewalling, etc. will be run as software on standard x86 servers, and are thus referred to as Virtualized Network Functions (VNFs). Note that VNFs may be executed directly by the hypervisor or within a Virtual Machine (VM) (or a container). In any case, to allow communications between the VMs (of a same physical server) and the rest of the physical network, the hypervisor can create a virtual switch (aka vSwitch) that logically connects the VMs to the outside world.

While software-based solutions are generally viewed as more flexible than their hardware counterparts, the network softwarization raises concerns about its expected performance. To address this issue, a consortium including companies like Intel and 6WIND has devised Data Plane Development Kit (DPDK) [2]. DPDK is an open-source project and works as a specialized library for x86, ARM and PowerPC processors. In particular, it enables vSwitches to accelerate the processing of incoming packets by (i) balancing the incoming flow of packets over all the vSwitch CPU cores, (ii) avoiding unnecessary re-copy of the packets, (iii) keeping all operations out of the OS kernel and, instead, within the user space and (iv) processing packets by batches, thereby having a better use of the CPU cache. While other libraries exist, DPDK has become a de facto standard for vSwitches. Nonetheless, due to the relative novelty and complexity of virtual switches, their performance modeling (e.g., to analytically derive estimates of throughput, loss rate, latency) remains a challenging problem. We believe that an analytical model can provide some helpful guidelines by suggesting adequate values for the large number of parameters that can be adjusted in a virtual switch.

In this paper, we investigate the performance of a virtual switch, i.e., a software relaying packets (possibly after modifying their content) between the ports of VMs or containers hosted on a same physical machine and the rest of the physical network. We assume that the vSwitch is equipped with DPDK. We propose a conceptually simple and easy-to-implement modeling approach for evaluating customary performance parameters of a vSwitch such as its mean throughput, its mean

latency, its loss rate as well as the level of utilization of its CPU cores. We consider several examples inspired by real scenarios and we assess our model accuracy by comparing its predictions with the actual values delivered by a discrete-event simulator. To illustrate the application of our model, we explore the effects of changing a vSwitch configuration (e.g., batch sizes) as well as adjusting the vSwitch resources (number of allocated CPU cores) to satisfy a given Service Level Agreement (SLA) policy (e.g., zero-loss).

The remainder of this paper is organized as follows. Section II describes the internal behavior of a DPDK-based vSwitch as well as the real-world inspired scenarios that motivate our study. In Section III, we present the main principles underlying our modeling framework for a vSwitch. Section IV details our approach in the case where a vSwitch processes incoming packets individually while Section V handles the case of packets being served by batches. We study the accuracy of our models in Section VI and we provide potential applications of theirs in Section VII. We discuss the related work in Section VIII. Section IX concludes this paper.

II. Description of a vSwitch

A. Context and definition

Like traditional switches, a vSwitch commutes packets between its ports but, unlike them, it operates as a software typically run by the hypervisor of the physical server. In general, a vSwitch has access to a set of logical CPU cores and to a set of physical and logical ports. Logical ports connect to ports of VMs hosted on the same physical machine while physical ports are associated with existing ports on the physical server. An example is given in Figure 1. For example, in a cloud computing context, a physical server may host several VMs that are interconnected using a vSwitch, which itself is executed by the hypervisor. In an SDN/NFV-based network, vSwitches (software running on commodity hardware) are replacing specialized hardware devices such as switches, firewalls, load-balancers, routers, and other middle-boxes. In addition to commuting packets between their ports, vSwitches may also perform other operations like filtering, header editing, content encrypting and deep packet inspection. In fact, vSwitches may use all the headers from layer 2 up to layer 5 (and not just 2 as it is commonly the case for a traditional switch) so that they are also referred to as virtual multi-layer switches. Note that the physical machine running the vSwitch typically hosts VMs or containers so that the vSwitch has to share the physical resources with them.

The two prominent solutions to create a vSwitch in a hypervisor are OVS [3] and FD.io VPP [4]. Note that both are open-source projects and that VPP is the open-source version of Cisco's Vector Packet Processing. Aside from open-source implementations, a couple of proprietary virtual switch solutions have been released; e.g., by Cisco (Nexus 1000V) and VMware (vSphere Distributed Switch).

A vSwitch mainly comprises three types of components: (i) network interface cards (NICs) that altogether provide a total of N input/output (I/O) ports, (ii) a set of (logical) CPU cores that are in charge of processing the packets coming from the different I/O ports, and (iii) memory to store packets waiting for their processing, be it from the CPU cores or from the NICs. As a side note, note that packets are moved across these components through PCI buses.

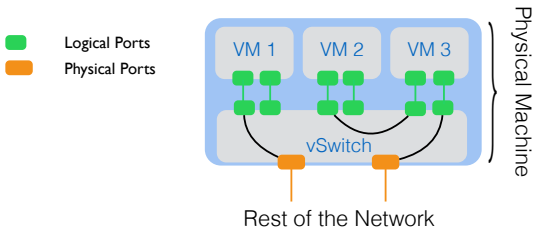


Fig. 1: vSwitch connecting three VMs with two physical ports.

B. DPDK library

To let vSwitches deal with high rates of packets, different techniques, e.g., Netmap [5], OpenOnload [6], PacketShader [7] and DPDK [2], have been developed to provide a faster packet processing. DPDK is an open-source project and works as a specialized library for x86, ARM and PowerPC processors. It is developed by a consortium comprising companies like Intel and 6WIND. Note that DPDK is integrated to the most prominent vSwitch solutions, namely OVS and VPP, making it a de facto standard for vSwitches. Hence, we focus our study on a vSwitch equipped with the DPDK library [2]. DPDK makes use of several means for accelerating the processing of packets.

1) No packets recopy: DPDK avoids the recopy of packets. Thanks to the use of a shared memory, CPU cores are able to process packets without recopying them in their associate memory. For a deeper understanding of these mechanisms, the reader can refer to the work of Scholz [8].

2) Operations performed in the user space: Another means of DPDK for accelerating the packet processing is to run the associated operations in the user space and not within the kernel as is done by default. By doing so, DPDK avoids the overhead of CPU interrupts that result in additional delays in processing packets.

3) Balancing the load across all the CPU cores: Although DPDK allows various configurations in the polling of the several vSwitch ports by the multiple CPU cores, we consider here its standard configuration, using the so-called "Poll Mode Driver", which is known as the most versatile and efficient (unless in specific scenarios). First, one CPU core, aka the "master" core, is entirely dedicated to the control and management of the vSwitch while the other CPU cores are devoted to the packet processing. Let C denote the number of CPU cores devoted to the packet processing, i.e., not including the master core. Second, DPDK aims at uniformly distributing the load originating from each port across the C CPU cores. Said differently, each CPU core contributes to processing packets coming from each port. More precisely, modern NICs perform load balancing by letting each of their ports dispatch incoming packets into C separate logical queues, called RX queues. This dispatching step is typically carried out through the application of a hash function on the packet headers (such

as the Receive Side Scaling (RSS) used in DPDK) and aims at ensuring an even balance of the incoming packets among the RX queues as well as at accelerating packet processing by directing packets belonging to the same flow to the same RX queue. As a result, each RX queue is assigned to a single CPU core while each CPU core handles as many RX queues as the total number of ports in the vSwitch. We denote by K the RX queue size, i.e., the maximum number of packets that can be queued simultaneously in it. Then, once a CPU core ends up processing a packet, the packet is (logically) forwarded from its RX queue to a TX queue associated to the appropriate output port. At this stage, the packet is pending for transmission on the next link and does not need any further CPU core processing resource. Figure 2 illustrates this mapping between CPU cores, ports and RX queues.

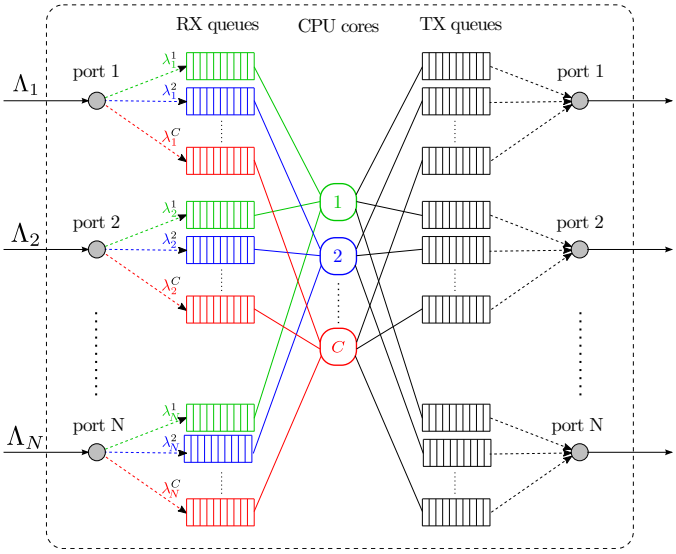


Fig. 2: Internal architecture of a vSwitch with N I/O ports and C CPU cores.

4) Processing packets by batches: CPU cores poll their associated RX queues in a cyclic order (i.e., in a round-robin fashion). However, for the sake of performance, DPDK enables CPU cores to serve a batch of packets on the same RX queue before switching to the next RX queue. We denote by T_S the mean switch-over time taken by a core to switch from its current RX queue to the next one, and by M the maximum size of the batch. When the batch size is set to M, a CPU core can prefetch up to M packets on one RX queue and then it processes them in a run-to-completion manner. Note that packets that enter the RX queue while the CPU core has already started its service are not served in this round, and they have to wait until the core revisits this queue. In the queueing theory literature, this discipline is known as a gated M-limited policy [9].

Batching packets by groups of M packets tends to increase the overall efficiency of a vSwitch. Indeed, when a CPU core handles packets belonging to the same batch, chances are that the needed instructions are found in the CPU cache, which lowers the average processing time of a packet. We denote by T_H and T_M the average time needed by a CPU core to process a packet when the set of instructions is found (cache Hit), respectively not found (cache Miss), in the cache. Note that T_H is typically significantly smaller than T_M, say around an order of magnitude or so. Let T_R indicate the average time needed by the CPU core to forward a packet from an RX queue to a TX queue over a PCI bus (once the CPU core processing has ended). As a result, the total time needed by a core to serve a given packet is either T_M + T_R (in case of a cache miss) or T_H + T_R (in case of a cache hit). In general, the former case is more likely to occur if the considered packet is among the first packets of a batch (the cache is likely to be "cold") whereas the latter has more chances to happen for the subsequent packets of a batch as they will benefit from the cache information. In addition, processing packets by batches also increases efficiency by reducing the total number of switch-over times (as a CPU core does not switch to a different RX queue upon the completion of a single packet processing).

Despite the enhancements brought by DPDK, vSwitches are subject to performance issues, in particular if the incoming load is too large. Given the transmission speed of lines and the current transfer rates of PCI buses and memory (SDRAM), the bottleneck of a vSwitch, if any, is likely to occur during the processing of packets in RX queues due to the limited CPU resources. Therefore, we concentrate our modeling efforts on the interactions between the CPU cores, the RX queues and the ports.

Note that, despite the "Poll Mode Driver" of DPDK wherein a master core is made available for the management tasks while the others are entirely devoted to the packet processing (see Section II-B3), a vSwitch may be affected by a momentary external load applied to the hypervisor or to co-hosted VMs. One way to take into account this effect is to scale up the average times to process a packet, T_H and T_M.

C. Scenarios

Throughout the paper, we consider three scenarios inspired by features of real vSwitches to demonstrate the accuracy and the abilities of our modeling approach. For the sake of simplicity and without loss of generality, we assume throughout this paper that the considered vSwitches comprise CPU cores running each at 3GHz, that RX queues are set to store up to $K = 128$ packets, that the mean packet size is 1000 bytes, and that the switch-over times T_S are equal to 1ns. We also assume that the dispatching function performed on incoming packets at the ports is well-behaved (see Section II-B3) so that every CPU core undergoes the same performance allowing us to restrict our analysis to only one of them. Note that this last assumption does not mean that ports are equally loaded. Note also that all the numerical values used to specify our three scenarios are derived from real-life experiments conducted in 6WIND lab.

1) Scenario 1 - Simple forwarding: In our first scenario, we consider a case where a vSwitch is simply forwarding incoming packets between its ports based on their link layer headers and does not provide any further services. Said differently, the vSwitch behaves similarly to a regular switch. We assume

that the vSwitch operates on a small-scale network so that its flow table is relatively small. More precisely, we assume that there is a total of $N = 4$ ports and that 80 CPU cycles are enough for processing packets (i.e., looking up entries in the flow table) while 10 additional CPU cycles (resp. 200) are needed to access the information if the vSwitch experiences a cache hit (resp. cache miss). Overall, given the speed of the CPU cores (3GHz), we have: $T_H = (80 + 10)/3 = 30$ns and $T_M = (80 + 200)/3 = 93.3$ns. We also assume that the batch size M is set to 4 packets and that PCI buses sustain 16 GBps so that $T_R = 1000/16 = 62.5$ns. Finally, we assume that ports are unevenly loaded as follows: Port 1 receives 15% of the whole traffic, Port 2 receives 20%, Port 3 receives 25% and Port 4 receives 40%.

2) Scenario 2 - Complex routing: Our second scenario pertains to a vSwitch whose flow table is large featuring numerous rules to handle different types of traffic with various destinations. Such a situation can occur for routers located in the core (backbone) network of a network operator. Here, we assume that, because of the size of the flow table, 800 CPU cycles are needed for the lookup operation while 10 additional CPU cycles (resp. 200) are needed to access the information if the vSwitch experiences a cache hit (resp. cache miss). Therefore, we obtain: $T_H = (800+10)/3 = 270$ns and $T_M = (800 + 200)/3 \simeq 333$ns. We choose a larger size of packet batch with $M = 8$ and we assume that PCI buses work at 32 GBps so that $T_R = 31.25$ns. Finally, we assume a total of $N = 5$ ports that are irregularly loaded as follows: Port 1 receives 10% of the whole traffic, Port 2 receives 15%, Port 3 receives 20%, Port 4 receives 25% and Port 5 receives 30%.

3) Scenario 3 - IPsec: In our last scenario, we consider a vSwitch applying IPsec encryption operations on incoming packets. Network architects typically deploy IPsec tunnels to provide security for data communication between pairs of distant nodes. The packets are encrypted at the ingress of the tunnel and decrypted at its egress using computationally intensive encryption algorithms implemented in IPsec. We assume that 8,000 CPU cycles are spent to perform the encryption process and that 10 additional CPU cycles (resp. 200) are needed to access the information if the vSwitch experiences a cache hit (resp. cache miss). Given the speed of CPU cores (3GHz), this leads to $T_H = (8000 + 10)/3 = 2670$ns and $T_M = (8000 + 200)/3 = 2733.3$ns. The size of batches is set to $M = 16$ packets. The speed of PCI bus is fixed to 8 GBps so that $T_R = 125$ns. The total number of ports is set to $N = 8$ ports that are unevenly loaded as follows: Port 1 receives 5% of the whole traffic, Port 2 receives 10%, Port 3 receives 15%, Port 4 receives 18%, Port 5 receives 22% and Port 6 receives 30%.

III. General modeling framework

A. System notation

We start this section by reminding the notation introduced so far. As stated in Section II, C denotes the total number of CPU cores devoted to the packet processing and N represents the number of ports attached to the vSwitch. As a results, the total number of RX queues of the vSwitch is equal to $N \times C$.

TABLE I: Principal notation.

Symbol	Description
C	Number of CPU cores devoted to the packet processing
N	Number of ports
K	Capacity of the RX queues
M	Size of packet batches
T_H	Average time needed by a CPU core to process a packet in case of a hit in the cache
T_M	Average time needed by a CPU core to process a packet in case of a miss in the cache
T_R	Average time needed by a CPU core to forward a packet to a TX queue
T_S	Average time needed by a CPU core to switch to the next RX queue
β	Switch-over rate
Λ_i	Packet arrival rate on port i, $i = 1, \ldots, N$
λ_i^j	Packet arrival rate dispatched to the j-th RX queue of port i, $j = 1, \ldots, C$ and $i = 1, \ldots, N$
Λ^j	Packet rate bound to the j-th CPU core, regardless of their incoming port ($j = 1, \ldots, C$)
μ_i^j	Service rate of the j-th CPU core when it is serving the i-th RX queue, $j = 1, \ldots, C$ and $i = 1, \ldots, N$
U^j	Utilization rate of the j-th CPU core, $j = 1, \ldots, C$
B_i	Blocking probability at port i, $i = 1, \ldots, N$
b_i	Loss rate at the entrance of queue i, $i = 1, \ldots, N$
$\bar{q}_i$	Average number of packets in queue i, $i = 1, \ldots, N$
$\bar{r}_i$	Average sojourn time in queue i, $i = 1, \ldots, N$

Each RX queue has a finite capacity expressed as a maximum of K packets. Each CPU core cyclically polls its associated RX queues and processes at most M packets from each RX queue before switching to the next one. M is referred to as the batch size. Let us also recall that the average time needed by a CPU core to process a packet is denoted by T_H in case of a hit in the cache, and by T_M in case of a miss. We use T_R to refer to the average time needed by a CPU core to forward a packet to a TX queue while we use T_S to denote the average time taken by a CPU core to switch from its current RX queue to the next one.

We use Λ_i to denote the packet arrival rate on port i ($i = 1, \ldots, N$) while λ_i^j refers to the rate of packets dispatched to the j-th RX queue of port i, and hence handled by the j-th core ($j = 1, \ldots, C$). It follows that $\Lambda_i = \sum_{j=1}^{C} \lambda_i^j$ and, assuming the hash function dispatches equally across the RX queues, we have: $\lambda_i^j = \frac{\Lambda_i}{C}$. Finally, we denote by Λ^j the total rate of packets bound to the j-th core CPU core, regardless of their incoming port ($j = 1, \ldots, C$) so that $\Lambda^j = \sum_{i=1}^{N} \lambda_i^j$.

For the sake of our modeling framework we let μ_i^j denote the service rate of the j-th core when it is serving the i-th RX queue, while β denotes the switch-over rate. Note that by definition, we have, $\beta = 1/T_S$. We detail later how μ_i^j can be derived from T_H, T_M and T_R.

Table I summarizes the principal notation used in this paper.

B. Performance parameters

The objective of this paper is to develop an accurate and scalable modeling framework to derive performance parameters of the vSwitch. These metrics may pertain to the

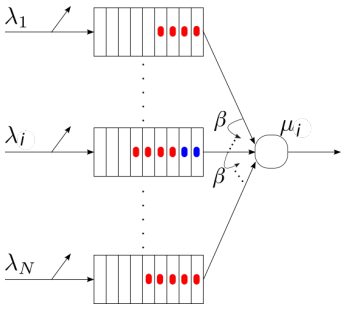


Fig. 3: Illustration of a subsystem involving a single CPU core that polls N RX queues with a size of batch of $M = 2$. Blue packets are being processed while red packets are waiting for their turn.

RX queues or to the whole system itself. As for the i-th RX queue ($i = 1, \ldots, N$) attached to the j-th CPU core ($j = 1, \ldots, C$), performance parameters of interest include the blocking probability (i.e., the loss rate) denoted by b_i^j, the mean sojourn time of a packet denoted by $\bar{r}_i^j$, as well as the buffer occupancy denoted by $\bar{q}_i^j$. Besides, for each CPU core j, we use U^j to indicate its utilization rate. Finally, the global performance of a vSwitch often derive from the former inner parameters. Thus, the global CPU core utilization rate, denoted by U, can simply be expressed as:

$$U = \frac{\sum_{j=1}^{C} U^j}{C} \tag{1}$$

Similarly, the packet blocking probability at port i, referred to as B_i, can be computed as:

$$B_i = \frac{\sum_{j=1}^{C} b_i^j \lambda_i^j}{\sum_{j=1}^{C} \lambda_i^j} = \frac{\sum_{j=1}^{C} b_i^j \lambda_i^j}{\Lambda_i} \tag{2}$$

Note that the global CPU utilization and blocking probability are performance parameters that capture and summarize the overall level of congestion in a vSwitch, and therefore represent metrics of direct interest for network operators.

C. Decomposition principle

The first step of the modeling framework is to break down the general switch architecture into C independent subsystems, each of them consisting of one CPU core that polls N independent RX queues. Recall that, as stated at the end of Section II-B, we focus on the interactions between the CPU cores, the RX queues and the ports, and as a result we exclude from the model the transmission part of packets in TX queues. Every subsystem is identified with a distinct color in Figure 2 and is simply referred to as a *polling system*. In the rest of the paper we only consider the model associated with a given CPU core j and its N related RX queues. Therefore, for the sake of clarity, we drop superscript j in all subsequent notations and equations. Figure 3 represents the polling system associated with the considered CPU core having a service rate μ_i when serving its i-th RX queue, and a switch-over rate β.

The second step of the general modeling framework consists in replacing each polling system with a set of N decoupled queues with server vacations. This decomposition step is illustrated in Figure 4. The buffer of queue i in the decomposed

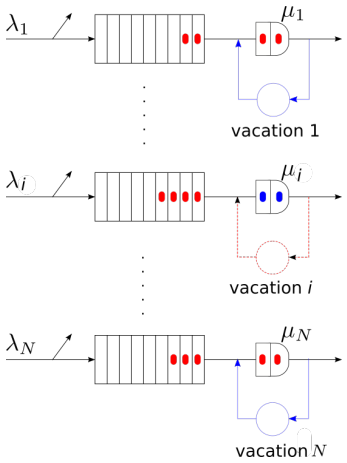


Fig. 4: Decomposition of a subsystem into N separate queues with a size of batch of $M = 2$. Blue vacations are active while the red vacation is inactive.

model represents the i-th RX queue associated with the considered CPU core. The server of the i-th queue in the decomposed model aims at reproducing the way packets of the i-th RX queue are processed by the CPU core. Because the core polls all its RX queues in-between the processing of two successive batches of (at most M) packets at a given queue i, there is an in-between time that corresponds to the processing of one batch of packets for all the other non-empty queues and N switch-over times. In the model, this total time will be referred to as a *vacation time*. As an illustration, in Figure 4, the server of queue i is in process, meaning that the CPU core is currently processing a packet of the current batch in RX queue i, and all other queues are in vacation. In this particular example, when queue i ends its processing, it goes in vacation, the remaining packets of queue i are put on a hold, and, at the same time, the switch-over time between RX queue i and RX queue $i + 1$ starts. It is only after the completion of this switch-over time that queue $i + 1$ ends its vacation and starts the processing of its first M in-line packets

D. Modeling assumptions

In order to derive tractable models, we make in the whole paper the following Markovian assumptions. First, we assume that the arrival of packets at the entrance of queue i follows a Poisson process of rate λ_i. Then, we assume that the processing time of one packet from queue i is exponentially distributed with rate μ_i. We also assume that the switching time is exponentially distributed with rate β. Finally, as developed in the following sections, the subsequent models will represent the vacation times by a succession of different phases with exponential distributions.

IV. Models with no batch services

To start our analysis, we do not take into account the processing of packets by batches and we restrict our study to the case of $M = 1$, i.e., a CPU core processes only one packet of its associated (non empty) RX queues at each round. We relax this assumption in the next section.

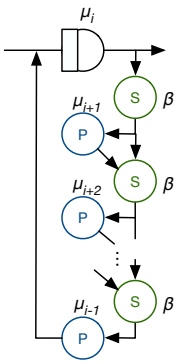


Fig. 5: Representing the vacation times in Sohail model [10].

With a batch size of $M = 1$, CPU cores are very unlikely to process consecutive packets belonging to the same flow, thereby essentially precluding any benefit from the cache. As a result the mean processing time of the considered CPU core when serving a packet from its i-th RX queue can be considered as a constant given by:

$$\frac{1}{\mu_i} = T_M + T_R. \tag{3}$$

The switch-over rate β is simply defined, as stated in previous section, as the inverse of the mean switch-over time:

$$\beta = \frac{1}{T_S}. \tag{4}$$

A. Sohail model

The model developed by Sohail in [10] falls into the scope of our general modeling framework. In their paper, the vacation time is developed as a sequence of switch-overs and possible processing times. As illustrated in Figure 5 in the case of queue i, its vacation time always comprises a first switch-over stage (first green "S" labeled circle) between RX queue i and RX queue $i+1$, and when this first switch-over stage ends, two things can happen: (i) if RX queue $i+1$ is not empty, the vacation time of queue i goes through the processing of a packet of RX queue $i+1$ (first blue "P" labeled circle), after which it continues with the second switch-over time between RX queue $i+1$ and RX queue $i+2$; (ii) if RX queue $i+1$ is empty, the vacation time of queue i goes directly to the second switch-over time. The same happens for all other RX queues so that the modeling of the vacation time becomes a complex phase-type distribution. As a result, the model associated with a given queue i is a pretty complicated Markov chain having $2N(K+1)$ states, that must be solved using a numerical technique [10]. Note that the parameters of the i-th Markov chain depend on the performance extracted from all other Markov chains, and thus an overall fixed-point iterative technique must be used to solve the model.

B. Proposed model

1) Vacation representation: Unlike Sohail model [10], in which the vacation time is represented as a sequence of switch-over and processing times, we keep in the vacation time the first switch-over time and aggregate all the remaining phases.

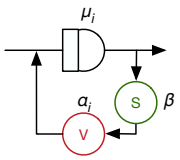


Fig. 6: Representing the vacation times in our model.

As shown in [11], by doing this we drastically reduce the complexity of the model without significantly deteriorating its accuracy. This idea is illustrated in Figure 6. Following the processing stage of the server, the vacation starts by a switch-over time between RX queue i and RX queue $i+1$. The remaining of the vacation time is then aggregated into a single exponential phase with a given rate α_i (i.e., with a given mean duration $1/\alpha_i$), that has to be accurately estimated.

2) Markov chain model associated with each RX queue: Under the markovian assumptions given in Section III-D, we can associate with each queue i of the decomposed model, the continuous-time Markov chain depicted in Figure 7. The chosen state description has two dimensions and thus is made of two parts $(k, \text{CPU state})$. The left-hand side corresponds to the current number of packets in the queue, $k = 1, \ldots, K$, while the right-hand side specifies if the CPU core is currently processing this queue (P), switching from this queue to the next one (S), or otherwise processing another RX queue or switching between the other RX queues (V).

In order to solve this chain corresponding to a particular queue i, only one parameter remains to be estimated, namely α_i (the other parameters λ_i, μ_i, β, and K are supposed to be known either from knowledge on the system or from measurements). However, assuming that α_i is known, we show in Appendix A how to quickly and easily obtain the stationary probabilities of this Markov chain without resorting to any numerical technique.

3) Estimation of the chain parameters: Instead of considering α_i, we estimate $1/\alpha_i$, corresponding to the mean time between the end of switching from i-th to $(i+1)$-th RX queue (marking the time when the core is leaving queue i) and the end of switching from $(i-1)$-th to i-th RX queue (marking the time when the core is returning to queue i). Therefore, this time includes $N-1$ switch-over times, but also includes the processing of one packet for all non-empty RX queue j different from i. It follows that:

$$\frac{1}{\alpha_i} = (N-1) \times \frac{1}{\beta} + \sum_{j \neq i} (1 - \mathcal{E}_j) \times \frac{1}{\mu_j}. \tag{5}$$

In this expression, $\mathcal{E}_j$ represents the probability that RX queue j is empty when the core is returning to it, i.e., at the particular instant when the switch-over time from $(j-1)$-th to j-th RX queue ends. This parameters can be extracted from the Markov chain associated with RX queue j (equivalent to the one represented in Figure 7 but where i is replaced by j). Indeed, $\mathcal{E}_j$ can be expressed as the ratio between the number of transitions from state $(0, V)$ to state $(0, S)$ by unit of time, and the total number of transitions from red states by unit of time, each of them correspondonding to the end of a vacation

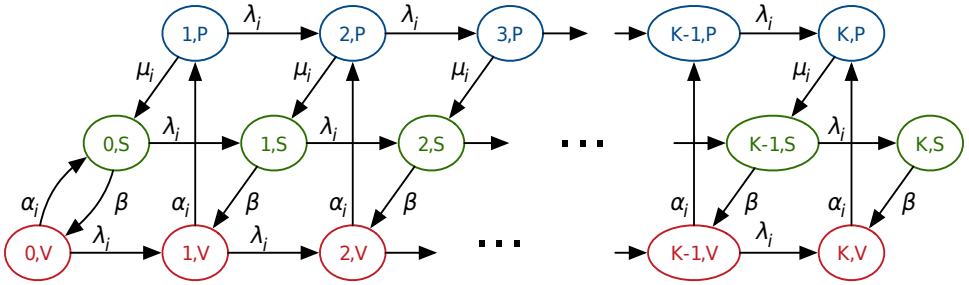


Fig. 7: Continuous-Time Markov Chain associated with queue i.

for RX queue j. Therefore, we have:

$$\mathcal{E}_j = \frac{\pi_j(0,V)\alpha_j}{\sum_{k=0}^{K}\pi_j(k,V)\alpha_j} = \frac{\pi_j(0,V)}{\sum_{k=0}^{K}\pi_j(k,V)}, \quad (6)$$

where the π_j are the stationary probabilities of the j-th Markov chain.

4) Fixed-point solution: As expected, the parameters of a Markov chain associated with a given queue i depend on the stationary solution of the other Markov chains (through Eq. 5 and 6). As a result, the resolution of the model relies on a fixed-point iterative technique that is summarized by Algorithm 1. The main loop of the algorithm is repeated until a given convergence criterion is reached, e.g., the maximum relative difference of varying parameters between two successive iterations is very small (say less that 10^{-7}).

Algorithm 1: Fixed-point iterative technique

```
Input  : System parameters K, μ_i, λ_i, β for each queue i
Output : Stationary probabilities π_i and performance metrics
         for each queue i
Initialize π_i, E_i for each queue i;
while convergence criterion not satisfied do
    foreach queue i ∈ [[1, N]] do
        Compute α_i using Eq. 5;
        Solve the Markov chain associated with queue i and
          compute the stationary probabilities π_i;
        Compute E_i using Eq. 6;
    end
end
Compute all performance metrics of interest from Eq. 7 to 9;
```

5) Performance parameters: After convergence of our algorithm, we can derive the system performance parameters from the stationary probabilities of the Markov chains as follows. The average number of packets in queue i is given by:

$$\bar{q}_i = \sum_{k=1}^{K} k \times (\pi_i(k,P) + \pi_i(k,S) + \pi_i(k,V)). \quad (7)$$

We can compute the loss rate at the entrance of queue i as:

$$b_i = \pi_i(K,P) + \pi_i(K,S) + \pi_i(K,V). \quad (8)$$

The average sojourn time of an accepted packet in queue i is then obtained using Little's law:

$$\bar{r}_i = \frac{\bar{q}_i}{\lambda_i(1-b_i)}. \quad (9)$$

V. Model taking into account batch services

We now turn to the more general case where packets are processed in batches by the CPU cores. As discussed in Section II-B, the use of batches accelerates the packet processing by reducing the number of CPU cores switch-over times, and more importantly, by making a better use of the CPU caches. Our rationale is to make use of the model presented in Section IV while carefully adjusting some of its parameters to account for the presence of batches.

A. Rethinking μ_i and β in the context of batch processing

In order to understand the influence of the cache on the mean processing times, let us take an example of a vSwitch having $N = 4$ ports and processing packets by batch of size $M = 16$. Let us consider that the CPU core at a particular round has to process a batch of 16 packets from RX queue 1 that is made of 4 packets destined to port 2, 7 packets destined to port 3, and 5 packets destined to port 4. We can reasonably assume that for the first packet destined to a given port, the information necessary for its processing (typically contained in the forwarding table) has to be fetched from the RAM, whereas for the subsequent packets destined to the same port, the information is present in the cache. If we make this assumption, the total time necessary to process all packets of the considered batch is thus $3T_M + 13T_H + 16T_R$, and the average processing time by packet is $\frac{3T_M+13T_H+16T_R}{16}$.

By generalizing this result, we first proposed in [12] to estimate the average processing time by the following simple equation:

$$\frac{1}{\mu_i} = \begin{cases} \frac{N-1}{M}T_M + \frac{M-(N-1)}{M}T_H + T_R & \text{if } M \geq N-1 \\ T_M + T_R & \text{if } M < N-1 \end{cases} \quad (10)$$

This estimation is realistic when $M >> N$ and when the system is heavily loaded. Indeed in this case there is a high chance that the CPU core processes full batches, i.e., batches made of M packets, and that in each batch there is at least one packet destined to each of the possible $N-1$ output ports. In this case, the processing time of the whole batch is $(N-1)T_M + (M-(N-1))T_H + MT_R$ and the formula becomes exact.

Finally, when considering a vSwitch with non negligible values of the switch-over time T_S, the value of β has to be adapted since the model does not structurally take batch services into account. The easiest way is to reduce the mean

switch-over time by a factor M, leading to replace Eq. 4 by [12]:

$$\beta = \frac{M}{T_S}. \tag{11}$$

B. Refining the calculation of μ_i

When M is small or when the load is low, Eq. 10 is not accurate anymore. Let us first see how we can improve the estimation of the mean processing time in the case where M can take any value (small or large). We first assume that a batch is always made of M packets (which will obviously be the case when the load is high). We propose to replace Eq. 10 by:

$$\frac{1}{\mu_i} = \frac{\bar{P}(M, N-1)}{M} T_M + \frac{M - \bar{P}(M, N-1)}{M} T_H + T_R \tag{12}$$

where $\bar{P}(M, N-1)$ is the average number of ports among the $N-1$ possible output ports that receive packets from the current batch (supposed of M packets):

$$\bar{P}(M, N-1) = \sum_{k=1}^{\min(N-1, M)} k P_k(M, N-1) \tag{13}$$

with $P_k(M, N-1)$ being the probability that a batch of M packets is sent over exactly k ports among the $N-1$ possible output ports, $k = 1, ..., \min(N-1, M)$.

If packets are assumed to be equally dispatched to all outgoing ports, $P_k(M, N-1)$ can be obtained through enumeration as:

$$P_k(M, N-1) = \frac{D_k(M, N-1)}{(N-1)^M} \tag{14}$$

where $D_k(m, n)$ is the number of ways to put m objects (the packets) in exactly k boxes (the output ports) among n possibles boxes. Theses quantities can be obtained from the following recursive formula:

$$\begin{cases} D_1(m, n) = n & k = 1 \\ D_k(m, n) = C_n^k k^m - \sum_{i=1}^{k-1} D_{k-i}(m, n) C_{n-(k-i)}^i & k = 2, ..., n \end{cases} \tag{15}$$

Let us now see how we can improve the value of the mean processing time $\frac{1}{\mu_i}$ in the case of a low load, i.e., when a batch is not anymore of M packets. If we first assume that we know the average size $\bar{m}_i$ of a batch on input port i. We propose to replace Eq. 12 by:

$$\frac{1}{\mu_i} = \alpha T_M + (1 - \alpha) T_H + T_R \tag{16}$$

where

$$\alpha = \begin{cases} \frac{\bar{P}(\bar{m}_i, N-1)}{\bar{m}_i} & \text{if } \bar{m}_i \text{ is integer} \\ (\lceil \bar{m}_i \rceil - \bar{m}_i) \frac{\bar{P}(\lfloor \bar{m}_i \rfloor, N-1)}{\lfloor \bar{m}_i \rfloor} & \\ \quad + (\bar{m}_i - \lfloor \bar{m}_i \rfloor) \frac{\bar{P}(\lceil \bar{m}_i \rceil, N-1)}{\lceil \bar{m}_i \rceil} & \text{if } \bar{m}_i \text{ is not integer} \\ & \text{and } \bar{m}_i > 1 \\ \bar{m}_i \bar{P}(1, N-1) & \text{if } \bar{m}_i \text{ is not integer} \\ & \text{and } 0 < \bar{m}_i < 1 \end{cases} \tag{17}$$

Now we have to face the problem of estimating the average size $\bar{m}_i$ of a batch.

This quantity can be easily obtained from the Markov chain as follows:

$$\bar{m}_i = \sum_{k=1}^{M-1} k \times \pi_i^{ev}(k) + M \times \sum_{k=M}^{K} \pi_i^{ev}(k) \tag{18}$$

where $\pi_i^{ev}(k)$ is the probability that queue i contains k packets at the instants of end of vacation of the queue ("ev"), i.e., when the precise size of a batch is decided. These probabilities can be derived from the stationary probabilities of the i-th Markov chain as:

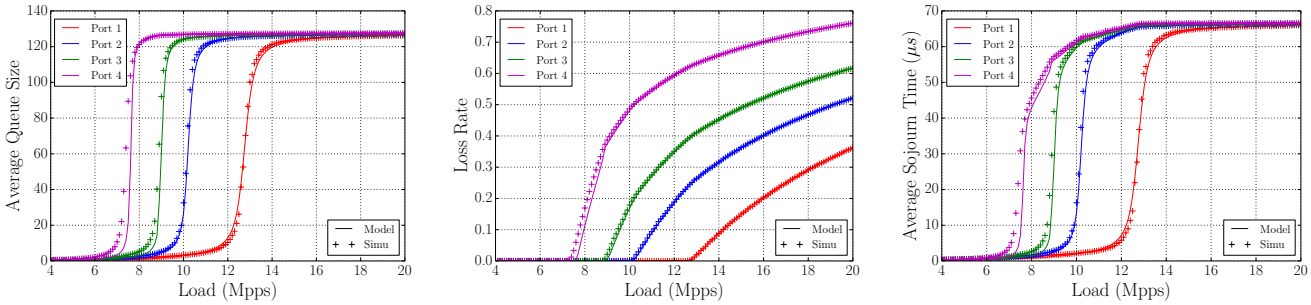
$$\pi_i^{ev}(k) = \frac{\pi_i(k, V)\alpha_i}{\sum_{j=0}^{K} \pi_i(j, V)\alpha_i} = \frac{\pi_i(k, V)}{\sum_{j=0}^{K} \pi_i(j, V)} \tag{19}$$

VI. Accuracy of the proposed approach

To study the accuracy of our modeling approach, we explore the three scenarios described in Section II-C. We compare the results of our model to those of a home-made discrete-event simulator written in Java, whose code is made available [13]. Unlike our model, the simulator precisely implements the behavior of a vSwitch as described in Section II: i) Packets queued on an RX queue are processed by batches; ii) The time to process a packet is closely related to the presence or absence of the corresponding instructions in the cache, and not only averaged as this is the case in the model; iii) After processing a batch of packets on a given RX queue, the CPU core is assigned to the next RX queue. As such, the simulator does not proceed, as the model does, with a decomposition of the vSwitch architecture into decoupled queueing subsystems with vacation. We provide a validation of our simulator against real-life measurements in Appendix B.

We run seven independent replications of 50,000,000 packets completions each. The obtained estimated confidence intervals at 95 percent confidence level are so small that we use only the mid-point in our validation. In each scenario, we consider a wide range of values for the packet arrival rate, varying from a very low level of load up to a high level corresponding to a full saturation of the vSwitch.

Numerical results for Scenario 1, in which we consider the case of a simple forwarding with $N = 4$ ports and batches of $M = 4$ packets, are presented in Figure 8. Refer to Section II-C for a complete list of the parameters. Figure 8a represents the average queue size (number of packets being buffered in RX queues) as a function of the load. We first notice that possible values of the average queue size vary from 0 for a low level of load up to 128 (corresponding to the capacity K of RX queues) when the load is high. As expected, we observe that the most loaded port, namely port 4 (receiving 40% of the total load), is the first to saturate when the load increases. For example, at a level of load of 8 Mpp, the queue associated with port 4 is almost full (i.e., close to 128 packets) while the queues associated with the three other ports are almost empty. Note also that the curves are significantly steep denoting a high sensibility to the actual level of load. In Figure 8b, we consider the loss rate (i.e., blocking probability) experienced by each port as a function of the load. We note



(a) Average queue size. (b) Loss rate. (c) Average sojourn time.

Fig. 8: Accuracy of our approach on Scenario 1 - Simple forwarding.

that the most loaded port is again the first to undergo losses as soon as the load exceeds 7 Mpps. Finally, we study the evolution of the average sojourn time spent by a packet in an RX queue in Figure 8c. At heavy levels of load, the value found for each port converges to a common asymptotic value (corresponding to the average time necessary to process 128 packets of a given RX queue). Overall, Figure 8 shows the close agreement between the proposed model and simulation.

Scenario 2 deals with the case of a vSwitch whose flow table is much larger and complex. Let us recall that, in this scenario, the number of ports is set to $N = 5$ while the size of batches equals $M = 8$ packets. Figure 9 presents the associated results. Figure 9c shows that, in the case of the average sojourn time, the asymptotic value for large levels of load differs from that found on Figure 8c. This gap results from the fact the mean time to process a packet is significantly larger in our second scenario. Here too, the performance parameters returned by our model closely match those delivered by the simulator.

Finally, Scenario 3, which addresses the case of a vSwtich performing IPsec functions and featuring $N = 6$ ports with a size of batches set to $M = 16$ packets, is handled in Figure 10. We again observe that the performance obtained from our model are close to those delivered from the simulator at any level of load.

VII. Examples of application

A. Influence of switch-over time

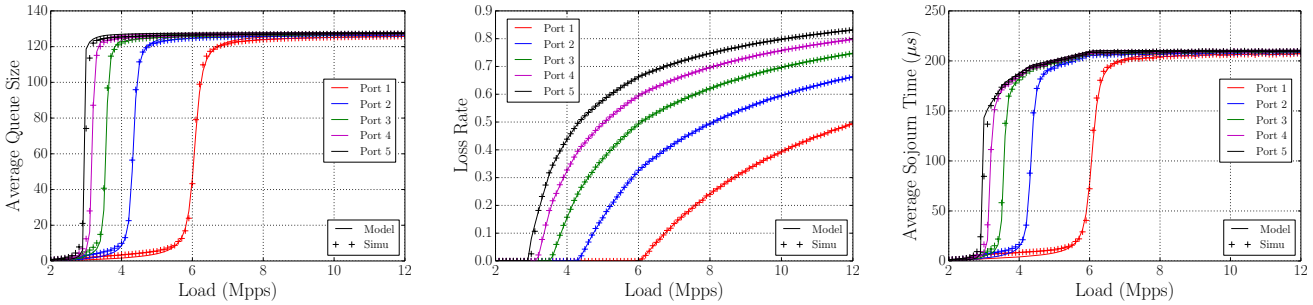
We begin by studying the impact of the switch-over time on the average size of RX queues. We use parameters close to those of Scenario 2 and let T_S vary so that it ranges from a very low overhead (i.e., representing 0.1% of the average packet processing time), all the way to a massive overhead (i.e., 100%). Then, based on our model, we compute the average number of packets buffered in the RX queue of the most loaded port for different levels of load. The associated results are reported in Figure 11. First, we notice that the relationship between T_S and the average queue size is far from being linear. Indeed, the deviation between an overhead of 100% and 50% is approximately twice smaller than that between 50% and 0.1%. Second, starting from a switch-over time representing approximately 2% of the packet processing time, all curves for subsequent smaller values tend to coincide. From a practical point of view, this suggests that, whenever the switch-over time represents an overhead less than, say 1% or 2%, they can be neglected in the performance analysis of a vSwitch.

We continue this study by examining simultaneously the influence of the switch-over time and of the load, on the loss rate of an RX queue. Figure 12 shows the corresponding results with a varying switch-over time on the X-axis and a varying load on the Y-axis. As shown by the figure, when the load is low (under 0.5-0.6 Mpps), the value of the switch-over time overhead has virtually no influence on the loss rate that remains totally negligible, whereas when the load is high (close to 1 Mpps), the switch-over time overhead has a large impact on the loss rate.

B. Influence of batch size

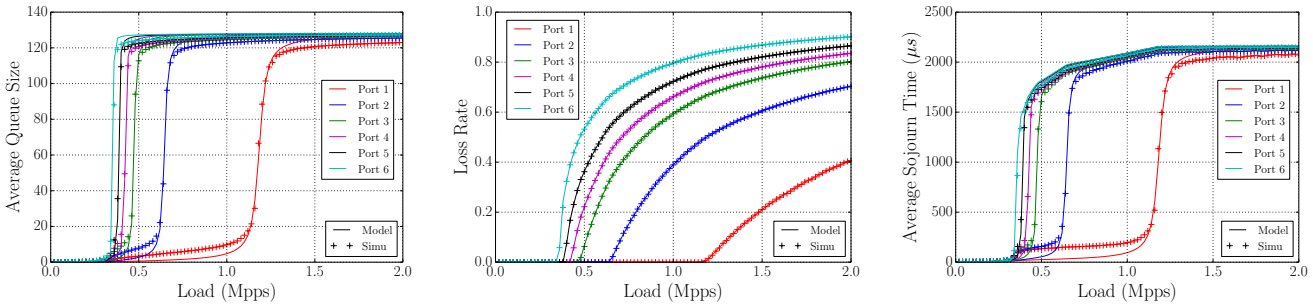
In our second example, we investigate the influence of the size of packet batches M on the performance of a vSwitch. We begin our study by considering Scenario 1. We let M vary from a value of 1, in which at most one packet of each RX queue is processed before the CPU core moves to the next RX queue, up to a value of 32. We restrict our analysis on the loss rate experienced by the most loaded port, namely port 4. Figure 13 shows the corresponding results. We observe that there is a substantial gain in increasing the size of packet batches as the onset of packet losses is postponed from a load of 6.5 Mpps for $M = 1$ to a load of almost 10 Mpps for $M = 32$, which represents an improvement of more than 50%. In the same way, while a load of 9 Mpps results into a severe congestion for a vSwitch parameterized with $M = 1$ (with a loss probability approaching 0.55), setting M to 32 leads to virtually no packets being lost. It is worth noting that the marginal gain of incrementing the size of batches decreases quite rapidly with growing values of M.

To develop a better understanding of the effect of M on the behavior of a vSwitch, we switch to Scenario 2 and we re-run our experiment. Let us recall that in this scenario the time for processing a packet varies much less between a cache hit and a cache miss. We represent the obtained results in Figure 14. As is shown, the size of the packet batches does not affect much the values obtained for the loss rates. Indeed, the gain obtained by increasing M from 1 to 32 barely reaches



(a) Average queue size. (b) Loss rate. (c) Average sojourn time.

Fig. 9: Accuracy of our approach on Scenario 2 - Complex routing.



(a) Average queue size. (b) Loss rate. (c) Average sojourn time.

Fig. 10: Accuracy of our approach on Scenario 3 - IPsec.

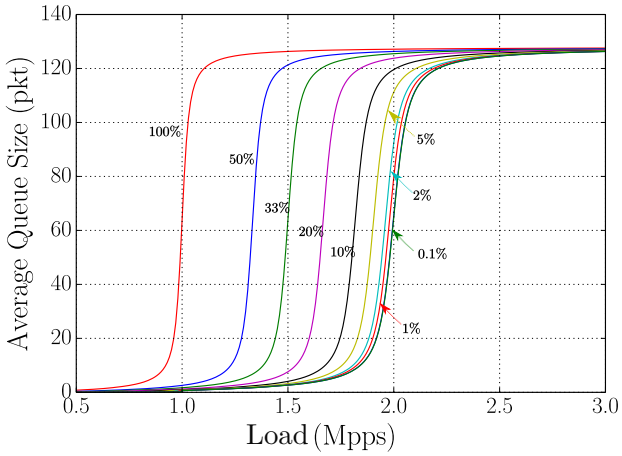


Fig. 11: Influence of the switch-over time on the average size of an RX queue for Scenario 2.

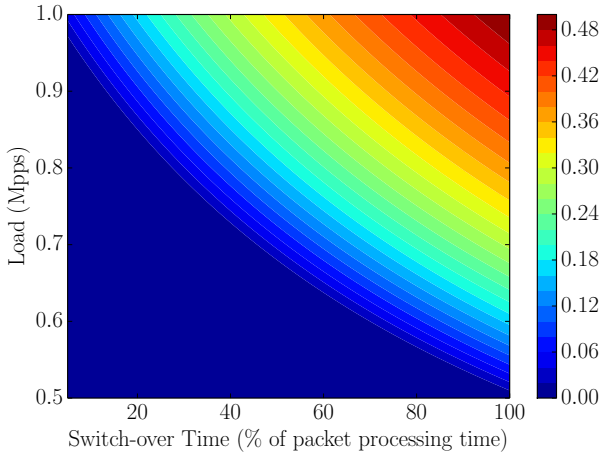


Fig. 12: Influence of the switch-over time and of the load on the loss rate experienced at an RX queue for Scenario 2.

10%. This is in stark contrast with Scenario 1. This difference stems from the nature of the load. Indeed, unlike Scenario 1, here incoming packets belonging to the same batch are quite unlikely to access the same information in the CPU cache, and hence there is little benefit in batching their services.

Based on these two examples, it appears that increasing the size of packet batches may significantly improve the performance of a vSwitch. However, the magnitude of the gain may vary widely depending on the characteristics of the packet processing times. Significant gains are expected when the average time needed to process a packet in case of a cache hit is significantly less than in the case of a cache miss.

C. *Ensuring zero-loss policy*

To illustrate the potential application of our model, we consider the problem of determining the proper number of CPU cores to meet a given QoS criterion. Operators often aim to size their networking devices so that packets exchanged over

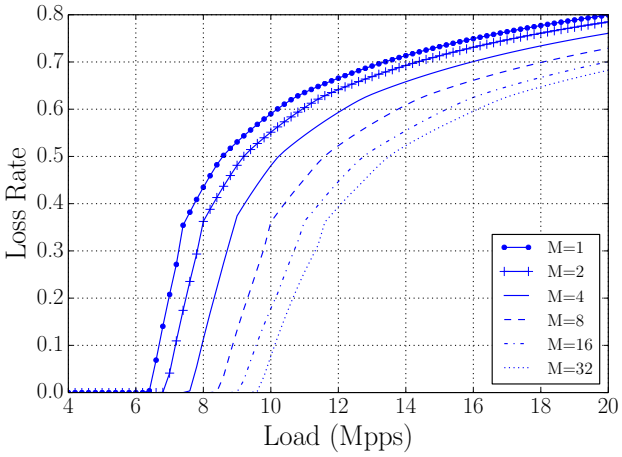


Fig. 13: Influence of the size of batches on the loss rate experienced at an RX queue for Scenario 1.

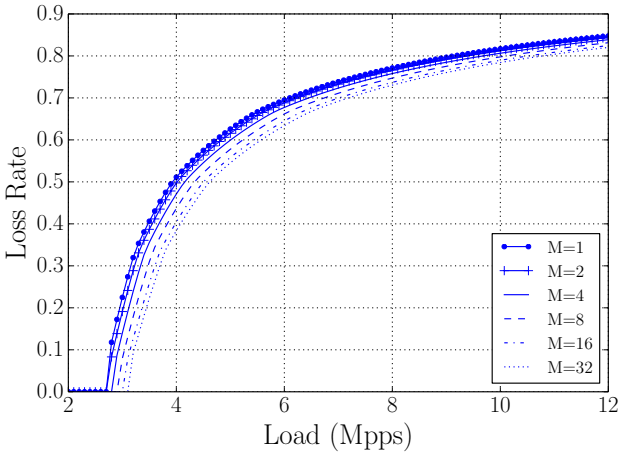


Fig. 14: Influence of the size of batches on the loss rate experienced at an RX queue for Scenario 2.

these devices are (almost) never dropped. We now describe how our model can help achieve this so-called "zero-loss" policy on a vSwitch.

We consider a vSwitch parameterized closely to that described in Scenario 3 but with $M = 1$. However, we let the number of allocated CPU cores, C, unspecified as it varies from 1 to 20. Then, for increasing values of the load submitted to the vSwitch, we compute the total loss rate experienced over all RX queues. Figure 15 shows the corresponding results. We observe that for a load of 8 Mpps, a minimum of $C = 10$ cores is required to ensure the zero-loss policy. This number grows to $C = 17$ cores if the load gets to 14 Mpps.

Dimensioning curves, analogous to those depicted in Figure 15, are easily and quickly delivered by our model. We believe that the knowledge conveyed by these curves can provide useful information in order to avoid the under- or over-provisioning of networking devices.

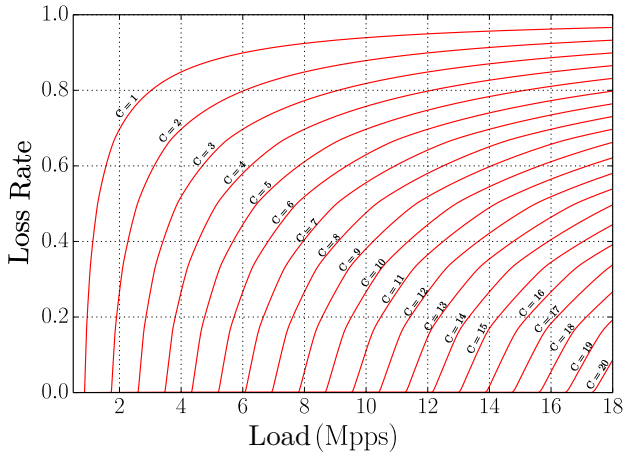


Fig. 15: Ensuring zero-loss policy by adequately determining the number of allocated CPU cores.

VIII. Related work

Many works have concentrated their efforts on evaluating the performance of DPDK-based vSwitch, either by conducting measurements or by using modeling techniques. In the following, we first present the approach adopted by DPDK's contributors to address this topic from an experimental point of view. We then review modeling attempts of vSwitch systems and, because polling is a key feature of DPDK systems, we study the related work of polling models. Finally, we conclude this section by reviewing our previous works on the performance evaluation of DPDK-based vSwitch.

A. Experimental approach

With the objective of achieving the best software switching performance, engineers contributing to the DPDK library developed internal performance measurement tools similar to monitoring probes. This approach has the advantage of giving a localized and precise analysis of low-level performance of a DPDK system, thus allowing developers to benchmark and validate any novel implementation or algorithmic optimization.

Nonetheless, the fact of reading and updating values in the system, typically when handling a counter, blocks the CPU cores for a while and so, reduces the overall system performance. About this, Vyatta/Brocade principal software architect stated during the 2015 Dublin DPDK Userspace forum that the act of "observing performance slows it down" [14] To overcome this issue the performances of DPDK based switching solutions are mainly assessed experimentally using dedicated test software such as the TestPMD application provided by Intel [15]. Such applications are for instance used by Intel to publish bi-annual DPDK performance reports [16]. Several works also attempted to experimentally characterize the open vSwitch performance. Most of them conducted measurements on an experimental testbed to demonstrate the enhancement provided by DPDK [17]. They also measured the impact of the number of NIC, the offered load, and the packet size [18]. Other works investigated the impact of active

flow monitoring [19], that might simply consist in sampling packets being forwarded across the vSwitch. Again, increasing the sampling rate to gain accuracy keeps consuming CPU resources, and in turn degrades the overall performance.

B. Modeling approach

By being non-intrusive in nature, modeling approaches can be used to overcome these measurement constraints and help in the performance evaluation of a DPDK system.

1) Non-DPDK-specific vSwitch modeling: Modeling the performance of vSwitch systems has been addressed for non-DPDK-specific systems. For instance, a model for estimating the packet loss probability and the average sojourn time of OpenFlow architectures is provided in [20]. This model assumes that all the packets arrive at the same queue before being forwarded to the switch. Suksomboon et al. presented an optimal configuration selection algorithm for Linux-based software routers relying on an Erlang-k-based packet latency prediction model [21]. Such prediction model uses measurements performed on two different router configurations to accurately predict the performance of all the configurations. It considers the case of systems with only a single CPU core.

These works cannot capture the effect of more sophisticated processing strategies such as polling used in the DPDK Poll Mode Driver or packet batch processing, that both highly contributes to a vSwitch performance enhancement [22].

2) Polling system modeling: As polling is a key feature of DPDK systems, we review the works done in modeling its performance. Strategies of polling have been extensively used in computer networks and telecommunication systems. For instance, the IEEE 802.5 Token Ring introduced in the early 1980s, used this scheme in its medium access method. Nonetheless, the analysis of polling systems started even before in the late 1950s with the patrolling repairman model for the British cotton industry. Reference surveys on polling models were published in the early 1990s by Takagi to provide a classification of polling systems and related research advances [23], [24]. These studies underscore that the performance of a polling system depends, in general, on many factors including the number and the capacity of queues, the arrival and service rates, as well as the switch-over time. Polling systems can be classified according to service policies, that might be exhaustive or gated, and unlimited or M-limited. Exhaustive: Once the server polls a given queue, it serves the queue until its complete exhaustion, and then it switches to the next queue. This implies that any request arriving in a queue while the server is currently processing another request of the same queue will be served before the server moves to the next queue. Gated: Unlike the exhaustive policy, the server does not process (in the current round) requests that may enter the queue while the server is already serving this same queue. Additionally, for both aforementioned policies, one can set an upper limit on the number of requests that the server can process for the same queue before switching to the next one. M-Limited: On each turn, the server can serve at most M requests for each queue. This corresponds to the case studied in this paper. Unfortunately, the general solution to polling systems is not known. However, their analysis is no less important, and therefore, several approximations have been developed. Tran-Gia proposed an analytical framework for computing the performance of a gated 1-limited polling system with non-zero switch-over time [25]. The modeling approach consists of solving a fixed-point problem to evaluate the state probabilities of an embedded Markov chain. In particular, the analysis of each involved queue is carried out at polling instants, i.e., ends of vacation. It requires the computation of Laplace-Stieltjes transforms as well as the use of Laplace inversion procedures or two-moment approximation techniques. As stated by the authors, such model is accurate only for large switch-over times and small values of the queue capacities (less than 10 requests). The fixed-point approach developed by Tran-Gia has then been extended to the case of exhaustive M-limited systems in [26]. In this work, the authors leverage the techniques provided by Lee to study *M/G/1/K* queues with server vacation [27], [28]. It consists in decomposing the polling system in individual *M/G/1/K* queues with server vacation. Each queue is then studied at polling instants. To reduce the number of modeling assumptions introduced in the previous works, a more general framework is presented in [29]. When conducting the analysis of each queue, it eliminates the hypothesis that the busy period, i.e., the time the server is not on vacation and that the vacation times are independent. This approach relies on solving a system of several numerical equations. However, as stated by the author, the complexity of the involved expressions may require using a symbolic computation software.

In conclusion, most of these approaches address a different policy than that implemented in vSwitches, and/or they involve complex arithmetic operations that may not scale with the number of queues or with their capacity. Although they accurately solve specific polling modeling problems, they seem to be of little help when evaluating the performance of general DPDK-based vSwitch systems.

3) Our previous contributions on DPDK-based vSwitch: In a 2016 paper, Artero et al. [30] described an analytical model for vSwitches based on the decomposition of the switch into several polling systems, each one being in turn decomposed into simple Markov chain models. Presented as a first step towards a more general model, this work did not take into account batch services and assumed a negligible switch-over time. Su et al. developed an extension of this work [11] that kept the simplicity and the accuracy of the initial model while taking into account switch-over times. Including batch services was all the more challenging as it structurally changes the nature of the models and drastically increases their complexity. A first attempt has been made in [12]. By simply adjusting the average processing times to reflect the cache effects, Su et al. have developed an extension of their previous works that provides an accurate approximation in the specific case of a heavily loaded system with large values of the batch size. The main challenge of developing a more general modeling framework including all the key features of DPDK-based vSwitches while adapting to a widest range of scenarios, remains an open issue and is the purpose of the present paper.

IX. Conclusions

In this paper, we present an analytical queueing model to evaluate the performance of a DPDK-based virtual switch with several CPU cores and network interface cards. Polling systems, in which a set of servers sequentially poll packets from a set of queues, with batch services, in which several packets are processed simultaneously, arises as an appropriate representation for modeling the behavior of a vSwitch. To circumvent the combinatorial growth of the state space associated with these models and their inherent complexity, we decouple the polling system associated with each CPU into several queues and we resort to servers with vacation to capture the interactions between queues. Our proposed solution is conceptually simple, easy to implement and computationally efficient.

We conduct tens of examples to assess the accuracy of our proposed model for various performance parameters such as the attained throughput, the packet latency, the buffer occupancy and the packet loss rate under various levels of loads. Comparisons against a discrete-event simulator show that our models typically deliver accurate estimates of the performance parameters. We illustrate how our models can help in determining an adequate setting of the vSwitch parameters using three real-life case studies, and derive some qualitative conclusions. For example, we find that increasing the size of packet batches may significantly improve the performance of a vSwitch, but only if a cache miss implies a much larger access time than a cache hit. Future works could aim at extending the validation of our model against real-life measurements.

Appendix

A. Efficient solution to the Markov Chain

The continuous-time Markov chain associated with a single queue i of the model of Section IV-B is illustrated in Figure 16 with some cuts that will be useful in the resolution.

Fist, let us assume that $\pi_i(0,V)$ is known. From the steady-state equation corresponding to cut C_0, we can express $\pi_i(0,S)$ as a function of $\pi_i(0,V)$:

$$\pi_i(0,S) = \frac{(\alpha_i + \lambda_i)}{\beta}\pi_i(0,V).$$

Then, from cut $C_0^{'}$, we derive $\pi_i(1,P)$:

$$\pi_i(1,P) = \frac{\lambda_i}{\mu_i}(\pi_i(0,V) + \pi_i(0,S)).$$

The three cuts C_1, $C_1^{'}$ and $C_1^{''}$, directly provide the three stationary probabilities $\pi_i(1,V)$, $\pi_i(1,S)$ and $\pi_i(2,P)$:

$$\pi_i(1,V) = \frac{\lambda_i}{\alpha_i}(\pi_i(0,V) + \pi_i(0,S) + \pi_i(1,P)),$$

$$\pi_i(1,S) = \frac{\lambda_i}{\beta}(\pi_i(1,V) + \pi_i(0,S) + \pi_i(1,P)),$$

$$\pi_i(2,P) = \frac{\lambda_i}{\mu_i}(\pi_i(1,V) + \pi_i(1,S) + \pi_i(1,P)).$$

Similarly, we can obtain the following stationary probabilities for any $k = 1, ..., K-1$:

$$\pi_i(k,V) = \frac{\lambda_i}{\alpha_i}(\pi_i(k-1,V) + \pi_i(k-1,S) + \pi_i(k,P)),$$

$$\pi_i(k,S) = \frac{\lambda_i}{\beta}(\pi_i(k,V) + \pi_i(k-1,S) + \pi_i(k,P)),$$

$$\pi_i(k+1,P) = \frac{\lambda_i}{\mu_i}(\pi_i(k,V) + \pi_i(k,S) + \pi_i(k,P)).$$

We use cuts C_K and $C_K^{'}$ to express $\pi_i(K,V)$ and $\pi_i(K,S)$:

$$\pi_i(K,V) = \frac{\lambda_i}{\alpha_i}(\pi_i(K-1,V) + \pi_i(K-1,S)),$$

$$\pi_i(K,S) = \frac{\lambda_i}{\beta}\pi_i(K-1,S).$$

Finally, the last unknown probability, $\pi_i(0,V)$, is obtained using the normalization constraint:

$$\sum_{k=0}^{K}(\pi_i(k,V) + \pi_i(k,S)) + \sum_{k=1}^{K}\pi_i(k,P) = 1.$$

B. Validating our simulator against real-life measurements

For a given scenario, we compare the results provided by our discrete-event simulator [13] that implements the behavior of a vSwitch as described in Section II with those collected on a real-life vSwitch implementing DPDK. The scenario is defined as follows. We consider a vSwitch with a total of $N = 32$ ports (equally loaded), RX queues of capacity $K = 128$ packets, CPU cores running at 2.3 GHz, a switch-over times T_S of 1ns, packet batch of size $M = 8$ and an average packet processing time of 178 CPU cycles corresponding to 77.43ns. The real-life vSwitch implements OVS with 6WINDGate and DPDK running on Ubuntu Linux with Intel Core i7 CPUs and Intel ixgbe NICs. Packets of 64 bytes were generated using IXIA. Figure 17 represents the corresponding results for the loss rate and the average queue size for a wide range of values of load. We observe that the results delivered by our simulator are in fair agreement with the experimental measurements.

Acknowledgments

This work was partly funded by the French ANR REFLEXION under the "ANR-14-CE28-0019" project. The authors thank Zidong Su for his help in implementing the simulator and Amine Kherbouche for his assistance in the definition of the scenarios. The authors also sincerely thank the anonymous referees for their thorough and constructive review that were of great help to improve this paper.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner, "OpenFlow: enabling innovation in campus networks," *Computer Communication Review*, vol. 38, no. 2, 2008.

[2] Data Plane Development Kit (DPDK). http://dpdk.org, 2017. Intel, 6WIND, etc.

[3] Open vSwitch - An Open Virtual Switch. http://www.openvswitch.org, 2017.

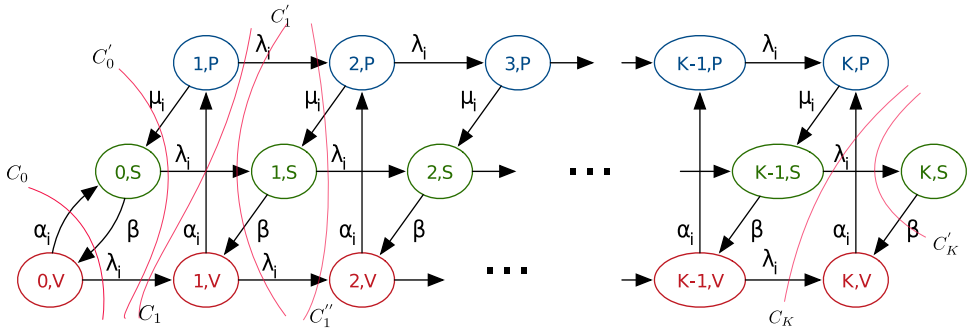
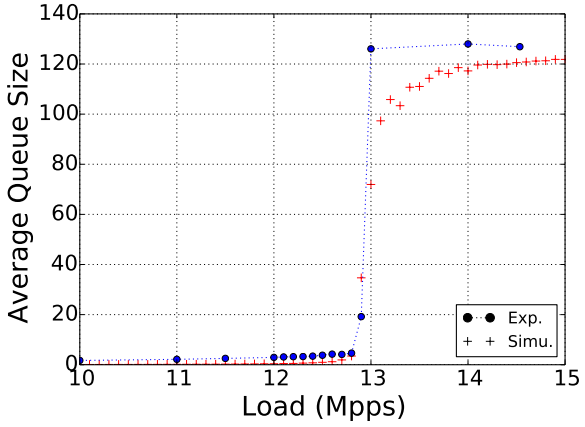
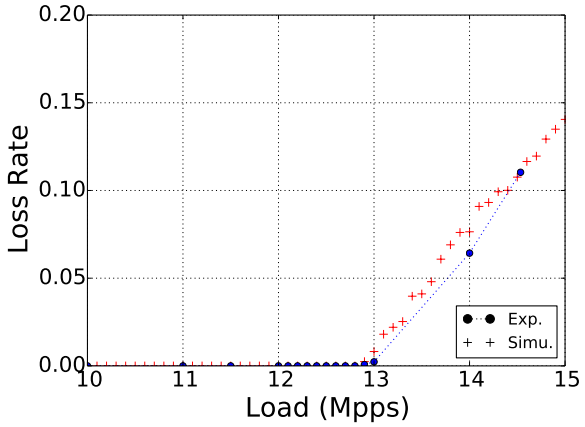


Fig. 16: Continuous-Time Markov Chain associated with queue i.



(a) Average queue size.



(b) Loss rate.

Fig. 17: Simulator results against real-life measurements.

[4] Fast data – Input/Output - Vector Packet Processing. https://wiki.fd.io/, 2017.

[5] L. Rizzo, "netmap: A novel framework for fast packet I/O," in *2012 USENIX Annual Technical Conference, June 13-15, 2012*, 2012.

[6] OpenOnload. http://www.openload.org, 2017. Solarflare.

[7] PacketShader. http://shader.kaist.edu/packetshader/, 2011.

[8] D. Scholz, "A look at intel's dataplane development kit," *Network*, vol. 115, 2014.

[9] H. Levy and M. Sidi, "Polling systems: applications, modeling, and optimization," *IEEE Transactions on communications*, vol. 38, no. 10, 1990.

[10] A. Sohail, "Performance evaluation of a non-exhaustive polling system with asymmetrical finite queues," in *14th International Conference on Computer Modelling and Simulation, UKSim*, 2012.

[11] Z. Su, B. Baynat, and T. Begin, "A new model for dpdk-based virtual switches," in *IEEE Conference on Network Softwarization (IEEE NetSoft 2017)*, 2017.

[12] Z. Su, T. Begin, and B. Baynat, "Towards including batch services in models for dpdk-based virtual switches," in *IEEE Global Information Infrastructure and Networking Symposium (IEEE GIIS 2017)*, 2017.

[13] DPDKSim - A simulator for DPDK-based Virtual Switches. https://github.com/garterog/DPDKSim, 2018.

[14] S. Hemminger, "DPDK performance, Lessons learned in vRouter," in *DPDK Summit, Userspace 2015, https://dpdksummit.com/Archive/pdf/2015Userspace/DPDK-Userspace2015-StephenHemminger-Performance.pdf*, 2015.

[15] Intel DPDK, "The TestPMD Application," in *http://dpdk.org/doc/guides/testpmd_app_ug/index.html*.

[16] Intel DPDK Validation team, "DPDK Intel NIC Performance Report Release 18.02, March 2018," in *http://fast.dpdk.org/doc/perf/DPDK_18_02_Intel_NIC_performance_report.pdf*.

[17] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, 2014.

[18] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with cuckooswitch," in *ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, ACM, 2013.

[19] V. Mann, A. Vishnoi, and S. Bidkar, "Living on the edge: Monitoring network flows at the edge in cloud data centers," in *International Conference on Communication Systems and Networks (COMSNETS)*, 2013.

[20] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an openflow architecture," in *2011 23rd International Teletraffic Congress (ITC)*, 2011.

[21] K. Suksomboon, N. Matsumoto, S. Okamoto, M. Hayashi, and Y. Ji, "Erlang-k-based packet latency prediction model for optimal configuration of software routers," in *2017 IEEE Conference on Network Softwarization (NetSoft)*, 2017.

[22] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet io," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, IEEE Computer Society, 2015.

[23] H. Takagi, "Queuing analysis of polling models," *ACM Comput. Surv.*, vol. 20, no. 1, 1988.

[24] H. Takagi, "Analysis of finite-capacity polling systems," *Advances in Applied Probability*, 1991.

[25] P. Tran-Gia and T. Raith, "Performance analysis of finite capacity polling systems with nonexhaustive service," *Performance Evaluation*, vol. 9, no. 1, 1988.

[26] M. Lang and M. Bosch, "Performance analysis of finite capacity polling systems with limited-m service," *13rd International Teletraffic Congress, ITC 1991*.

[27] T. T. Lee, "M/G/1/N queue with vacation time and exhaustive service discipline," *Operations Research*, vol. 32, no. 4, 1984.

[28] T. T. Lee, "M/G/1/N queue with vacation time and limited service discipline," *Performance Evaluation*, vol. 9, no. 3, 1989.

[29] D. Kofman, "Block probabilities, throughput and waiting time in finite capacity polling systems," *Queueing systems*, 1993.

[30] G. A. Gallardo, B. Baynat, and T. Begin, "Performance modeling of virtual switching systems," in *IEEE International Conference on Modeling, Analysis, and Simulation on Computer and Telecommunication Systems, MASCOTS*, 2016.