Supporting Fine-Grained Network Functions through Intel DPDK

Supporting Fine-Grained Network Functions through Intel DPDK

Ivano Cerrato, Mauro Annarumma, Fulvio Risso
Department of Computer and Control Engineering
Politecnico di Torino
Torino, 10129, Italy
Email: {ivano.cerrato, fulvio.risso}@polito.it
mauro.annarumma@studenti.polito.it

***Abstract*—Network Functions Virtualization (NFV) aims to transform network functions into software images, executed on standard, high-volume hardware. This paper focuses on the case in which a massive number of (tiny) network function instances are executed simultaneously on the same server and presents our experience in the design of the components that move the traffic across those functions, based on the primitives offered by the Intel DPDK framework. This paper proposes different possible architectures, it characterizes the resulting implementations, and it evaluates their applicability under different constraints.**

***Keywords*-DPDK; NFV; fine-grained network functions**

I. Introduction

Network Functions Virtualization (NFV) is a new network paradigm that proposes to implement in software the many network functions (NFs) that today run on proprietary hardware or dedicated appliances, by exploiting high-volume standard servers (e.g., Intel-based blades) and IT virtualization. This approach allows to consolidate several NFs (e.g., NAT, firewall, etc.) on the same machine; moreover, it offers an unprecedented agility to network operators, as the network service can be reconfigured dynamically by simply instantiating new NFs on the flight.

The most notable difference between classical IT virtualization and NFV is the degree of network traffic that has to be handled within a single server, as traditional virtualization has to deal mostly with *compute-bounded* tasks, while *network I/O* represents the dominant factor in the NFV case. This poses non trivial challenges when writing both NFs and the system framework (e.g., the hypervisor), as many low-level details such as memory access patterns, cache locality, task allocation across different CPU cores, synchronization primitives and more, may have a dramatic impact on the overall performance.

To facilitate the development of network I/O-intensive applications, Intel has recently proposed the Data Plane Development Kit (DPDK) [1], a framework that offers efficient implementations for a wide set of common functions such as NIC packet input/output, easy access to hardware features (e.g., SR-IOV, FDIR, etc.), memory allocation and queuing.

In this work we exploit the primitives offered by DPDK to investigate the case in which a huge number of NFs are executed simultaneously on the same server, e.g., when the network is partitioned among several tenants (potentially individual users), each one having a set of NFs that operate only on the traffic of the tenant itself. This requires the system to execute many NF *instances*, leading to a situation in which thousands of NFs may be running on the same server, although each instance will be characterized by a minuscule workload.

This paper focuses on the design of the components that deliver (and receive back) the traffic to the NFs, one of the topics of the EU-funded FP7 project UNIFY [2], which aims at providing full network and service virtualization to enable rich and flexible services. In particular, efficient data exchange is one of the requirements mentioned in the deliverable D5.1, which provides the functional specification of a platform that is potentially able to deliver both computing and network virtualized services. This paper proposes several possible architectures, each one targeting a specific working condition, for transferring data between the virtual switch (shown in Figure 1) and NFs, exploiting (whenever possible) the primitives offered by the Intel DPDK framework. Our goals include the necessity to scale with the number of NFs running on the server, which means that we should ensure high throughput and low latency even in case of a massive number of NFs operating concurrently, each one potentially traversed by a limited amount of traffic.

The paper is structured as follows. Section II provides an overview of DPDK, while Section III describes the general architecture of the framework to (efficiently) provide traffic to a massive number of NFs. Several implementations of this framework are then provided in Section IV, while their performance are evaluated and compared in Section V. Section VI discusses related works, and finally Section VII concludes the paper and proposes some future extensions to this work.

II. DPDK overview

Intel DPDK is a software framework that offers to programmers a set of primitives that help to create efficient NFs on x86 platforms, in particular high speed data plane applications.

DPDK assumes that processes operate in polling mode in order to be more efficient [3] and reduce the time spent by a packet traveling in the server. This would require each process to occupy one full CPU core (in fact, DPDK processes are pinned to a specific CPU core for optimization reasons), hence the number of processes running concurrently are limited by the CPU architecture.

Although this scheduling model is not mandatory, DPDK primitives are definitely more appropriate when applications are designed in that way; for example, DPDK does not offer any interrupt-like mechanism to notify a NF for the arrival of a packet on the NIC.

DPDK supports multi-process applications, consisting of a primary process enabled to allocate resources such as `rte_ring` and `rte_mempools`, which are then shared among all the secondary processes. A DPDK process, in turn, consists of at least one *logical core (lcore)*, which is an application instance running on a CPU core.

To manage memory, DPDK offers the **`rte_malloc`** and the **`rte_mempool`**. The former looks similar to the standard *libc* `malloc`, and can be used to allocate objects *(i)* on huge pages (in order to reduce IOTLB misses), *(ii)* aligned with the cache line and *(iii)* on a particular NUMA socket in order to improve the performance of the applications. The **`rte_mempool`**, instead, is a set of pre-allocated objects that can be acquired, and later possibly released, by `lcores` according to their needs. Since the same `rte_mempool` can be shared across `lcores`, a per-core cache of free objects is available to improve performance. In addition to the performance techniques already mentioned with respect to the `rte_malloc`, all objects within the `rte_mempool` are aligned in order to balance the load across different memory channels. This is particularly useful if we always access the same portion of the object, such as the first 64B of packets.

To exchange data among each others, `lcores` can use the **`rte_ring`**, a lockless FIFO queue that allows burst/bulk-single/multi-enqueue/dequeue operations. Each slot of the `rte_ring` contains a pointer to an allocated object, hence allowing data to be moved across `lcores` in a zero-copy fashion. If the `rte_ring` is used to exchange network packets, each slot of the buffer points to an **`rte_mbuf`**, which is an object in the `rte_mempool` that contains a pointer to the packet plus some additional metadata (e.g., packet length).

Finally, the **Poll Mode Driver (PMD)** library is the part of DPDK used by applications to access the network interface cards (NICs) without the intermediation (and the overhead) of the operating system. In addition, it also allows applications to exploit features offered by the Intel NIC controllers, such as `RSS`, `FDIR`, `SR-IOV` and `VMDq`. The PMD does not generate any interrupt when packets are available in the NIC, hence the `lcores` that receives packets from the network should implement a polling model. As a final remark, packets received from the network are stored into a specific `rte_mempool`.

III. General architecture

The general architecture of our system is shown in Figure 1. A virtual switch (vSwitch) module (*i*) receives packets from both NICs and NFs, (*ii*) classifies and (*iii*) delivers them to the proper NF according to the service chain each packet belongs to. Finally, when a packet has been processed by all the NFs associated with its service chain, (*iv*) the vSwitch sends it back to the network.

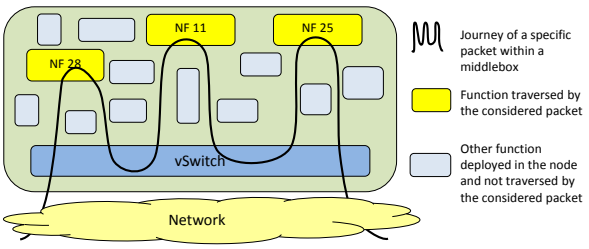


Figure 1. High-level view of a server with a vSwitch and several NFs.

The vSwitch is created at startup; instead, NFs are created at run-time as new tenants may be dynamically attached/detached to the network node. This implies the necessity to (*i*) dynamically create the proper set of NF instances associated with the new tenant and (*ii*) dynamically reconfigure the service chain for that tenant, which translates into setting the proper forwarding rules for the tenant's traffic in the vSwitch.

Our NFs are simple UNIX processes instead of full-fledged virtual machines as suggested in the NFV paradigm. This is needed because of (*i*) the massive amount of NFs that we need to handle (hence the pressure on CPU and memory occupancy of each function that would make VMs unpractical) and (*ii*) the startup time, as we cannot wait tens of seconds for a NF to be active. However this does not represent a limitation because we focus on the communication mechanism between the different components, which is orthogonal to the architecture of the components themselves.

The vSwitch is a simplified virtual switch that supports only forwarding rules based on MAC addresses; while this looks limiting compared to other equivalent components such as OpenvSwitch [4], it allows us to focus on the transmit/receive portions of the switch, limiting the overhead due to the presence of other features.

Finally, following the DPDK recommendations, the vSwitch operates in polling mode as it is supposed to process a huge amount of traffic (each packet traverses the vSwitch multiple times), while NFs may follow either the polling or interrupt-based model, depending on considerations that will be detailed in the following.

IV. Implementations

This section details five possible implementations of the architecture described in Section III. Each implementation is a multi-process DPDK application, where the vSwitch is the primary process (single `lcore`) and each NF is a different (single `lcore`) secondary process (except for those described in Section IV-E). This is required because of the necessity to create/destroy the processes containing NFs at run-time, while all the `lcores` of the primary process must be created at startup.

Unfortunately, vanilla DPDK does not support the execution of two different secondary processes on the same CPU core, which is a fundamental requirement in our use case, since we envision thousands of NFs deployed on the same physical server. To overcome this limitation,

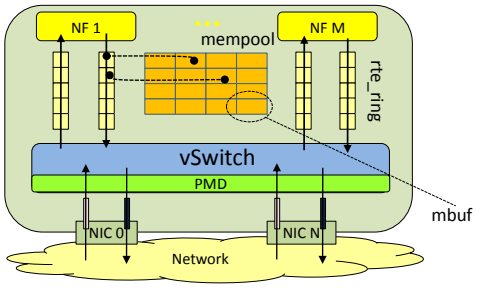


Figure 2. Implementation based on a (different) pair of rings shared between the vSwitch and each NF.

we modify the `lcore_id` internal DPDK variable in the initialization phase of each secondary process, so that each NF has its own DPDK internal data structures (and then no conflict can arise among NFs).

A. Double buffer

In this implementation (Figure 2) each network interface is configured with a single input and a single output queue; all the packets entering in the node are first processed by the vSwitch, which accesses to the NICs through the PMD library. Each NF exchanges packets with the vSwitch through a couple of `rte_rings`: one used for the communication vSwitch $\rightarrow$ NF, the other used for sending back to the vSwitch those packets already processed by the function itself. Finally, all the elements of the `rte_rings` points to `rte_membufs` in the same `rte_mempool`, allocated by the vSwitch at startup.

In this case NFs operate in polling mode, hence they never suspend spontaneously. Hence, this implementation is appropriate for those cases in which a limited number of NFs is active, even not higher than the number of CPU cores available on the server.

B. Double buffer + semaphore

In this second implementation, NFs operate in blocking mode: the vSwitch wakes up, through a POXIS named semaphore, a NF when a given number of packets is available for that application. When all the packets in the buffer have been processed, the NF suspends itself and waits for the next signal from the vSwitch. Obviously, this mechanism is complemented by a packet aging timeout that wakes up the NF if there are packets waiting for too long, hence avoiding data starvation.

This implementation is appropriate when NFs need to process a limited amount of traffic. In this case, the polling model would unnecessarily waste a huge amount of CPU resources, while a blocking model allows to increase the density of the NFs active on the same server. In fact, in this case a NF suspends itself when no packets are available, freeing the CPU that can be used by another NF that actually has packets to be processed.

C. Single buffer towards the vSwitch + semaphore

In the implementations described so far, the vSwitch may have to handle a huge number of "downstream" buffers coming from NFs, which may require a considerable amount of time while working in polling mode.

In this third implementation all NFs share a single "downstream" `rte_ring` toward the vSwitch, which exploits the lock-free multi-access capabilities of that structure. This would result in a saving of CPU cycles when iterating on a fewer `rte_rings`, as well as an improved cache effectiveness thanks to the better locality in memory access patterns.

This implementation *may* be appropriate when a large number of NFs are active on the same server, as we expect that in each one of its running rounds the vSwitch would find a few applications with packets ready to be consumed. Unfortunately, according to [1], the multi-producer enqueue function implemented in DPDK does not allow two or more NFs executed on the same CPU core to use the same `rte_ring`. Hence, although very appealing, this architecture has not been implemented because it would support only a limited number of NFs (less than the number of CPU cores).

D. Double buffer + FDIR

This forth implementation aims at reducing the load on the vSwitch, which is undoubtedly the most critical component of the system, by allowing some NFs to receive directly the traffic coming from the NICs.

For this, we use the FDIR (Flow Director) facility, which allows each NIC to be initialized with several input queues (incoming traffic is distributed based on the value of specified packet fields) and a single output queue. Each input queue is then associated with a different NF, while the output queue is just accessed by the vSwitch. When a NF is started, the vSwitch adds a new `FDIR` perfect filter on all the NICs, and binds this filter with a specific input queue of each port. This way, the first classification of packets is offloaded to the NIC, hence the vSwitch has just to move packets between NFs and send on the network the ones already processed by the entire service chain. However, this higher efficiency is paid with more complex NFs, which have to handle multiple input queues, namely the ones created by the NIC (accessed through the PMD) and the `rte_ring` shared with the vSwitch.

Since the number of hardware queues available on the NICs is limited, this architecture is appropriate when the number of NFs is reduced. Alternatively, if the number of NFs is huge, an hybrid architecture may be used: some NFs only receives traffic from the vSwitch, while others (which are at the beginning of the service chains) are directly connected to a queue of the NIC.

E. Isolated buffers + semaphore

The last implementation targets the case in which NFs are not trusted and hence we cannot allow them to share a portion of the memory space with the rest of system, as all the processes belonging to the same DPDK application share all the data structures created by the primary process (e.g., `rte_mempool`, `rte_rings`).

In this implementation only the vSwitch is a DPDK process, while each NF is a separated (non-DPDK) process.

This way, the `rte_mempool` containing traffic coming from the NICs can only be accessed by the vSwitch, which will provide each packet only to the proper NF. To this purpose, the vSwitch shares with each NF a distinct set of three buffers: two are similar to the DPDK `rte_rings`, and contain a reference to a slot in the third one, which is actually a simple memory pool containing only the packets exchanged between the vSwitch and the NF. This requires one additional copy each time a packet has to be delivered to the next NF in the chain, i.e., from the `rte_mempool` to the per-NF buffer when the packet has just be received from the NIC, and between those per-NF buffers in the next steps of the service chain.

Since in this implementation we cannot exploit DPDK-based functions neither in the NF, nor in the per-NF buffers, we had to implement (manually) all the techniques provided by the DPDK for efficient packet handling; among the others, buffers starting at a memory address that is multiple of the cache line size and storing each packet to an offset that is multiple of the cache line size.

This implementation aims at providing the adequate *traffic isolation* among NFs and is appropriate when an operator does not have the control on the NFs deployed on its network nodes, e.g., when tenants are allowed to install "opaque" NFs on the network, which are not trusted by the operator itself.

V. Performance evaluation

This section evaluates the performance of the implementations described in Section IV. Tests are executed on dual E5-2660 Xeon (eight cores plus hyperthreading) running at 2.20GHz, 32GB RAM and one Intel X540-based dual port 10Gbps Ethernet NIC. Two other machines are used respectively as a traffic generator and a traffic receiver, connected through dedicated 10Gbps Ethernet links.

Each test lasted 100 seconds and was repeated 10 times, then results are averaged. Each graph representing the maximum throughput is provided with a bars view that reports the throughput in millions of packets per second on the left-Y axis, and a points-based representation that reports the throughput in Gigabit per second on the right-Y axis. Instead, latency measurements are based on the `gettimeofday` Unix system call and include only the time spent by the packets in our system, without the time needed to actually send/receive data on the network.

Tests are repeated with packets of different sizes and with a growing number of running NFs; moreover, each packet traverses two of these NFs. Traffic is generated so that two consecutive packets coming from the network must be provided to two different NFs in order to stress more the system. The size of the buffers has been chosen in order to maximize the throughput of the system.

The NFs used in tests simply calculate a signature across the first 64 bytes of each packet, which represents a realistic workload as it emulates the fact that most network applications operate only on the first few bytes (i.e., the headers) of the packet in read-only mode.

Finally, unless otherwise specified, we used only the CPU whose socket is directly connected to the NIC.

A. Double buffer

Figure 3(a) shows the throughput achieved with a growing number of NFs deployed on the "double buffer" architecture. In particular, from the figure it is evident that the throughput is maximized when no more than one NF is executed on a physical core[1]. In fact, it drops of about 20% (with 64B packets) when the number of NFs changes from 7 to 8, due to the fact that we start allocating NFs on the logical cores of CPU0 as well, hence having multiple NFs that share the same physical core.

Figure 4(a) plots the latency experienced by packets in our system and shows that its value tends to increase considerably with the number of NFs, as shown by an average value of 24.44ms with 100 NFs.

This model looks appropriate only if the number of NFs is smaller than the number of CPU cores available; after that point the throughput drops and the latency becomes barely acceptable.

B. Double buffer + semaphore

Figure 3(b) depicts the throughput obtained with a growing number of NFs with the architecture described in Section IV-B. In particular, it shows that the NFs implemented in blocking mode achieve higher throughput than in the previous case, in which they operated in polling mode. This allows the system not only to go faster in any working condition (even when a few NFs are active), but to support an higher number of NFs without any significant drop in terms of performance, achieving just over 8Gbps with 700B packets even with 2000 NFs.

Interesting, the better throughput is not achieved at the expense of the latency, as shown in Figure 4(b). In fact, if with a few NFs we can assist to a negligible worsening (with 4 and 10 NFs, the average latency is less than 100μs higher compared to Figure 4(a)), things become rapidly far better with an higher number of NFs, achieving an average of 1,89ms and 4,83ms respectively with 40 and 100 NFs.

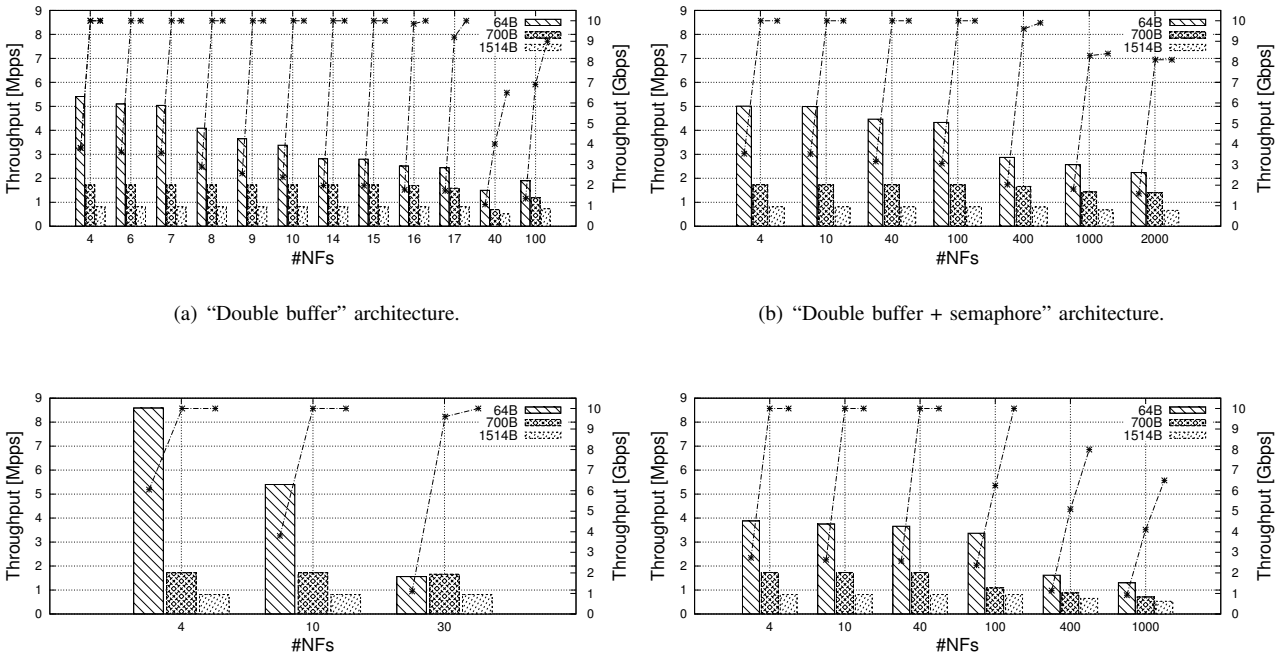
Finally, it is remarkable the fact that, with 2000 NFs, the 7 CPU cores allocated to them (the last is dedicated to the vSwitch) are loaded only at 18% in average, which shows the efficiency of the system. However, as evident, the latency is definitely not acceptable (an average of 160ms with 2000 NFs), that is the reason why we did not try to squeeze even more NFs on the system, although, from the point of view of the throughput, there was still room for more of them.

C. Double buffer + FDIR

Figure 3(c) shows the throughput achieved with the "double buffer + FDIR" implementation.

In this case we experienced a limitation of the DPDK framework: although the NIC controller exports 64 hardware queues (hence it can distribute the traffic to 64 different consumer processes), the DPDK forces each one of those processes (as part of a multi-process DPDK

[1]It is worth noting that, for performance reasons, one physical core is always dedicated to the vSwitch; hence, the machine used in the tests has still 7 physical cores (on CPU0) that can be assigned to NFs.



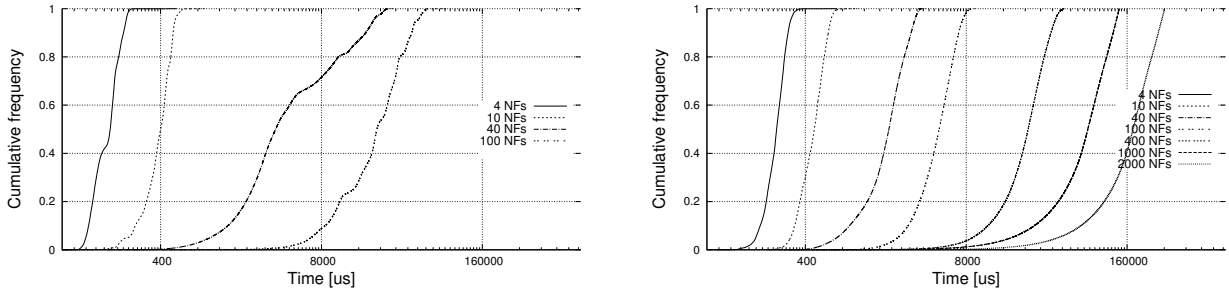
(a) "Double buffer" architecture.

(b) "Double buffer + semaphore" architecture.

(c) "Double buffer + FDIR" architecture.

(d) "Isolated buffers + semaphore" architecture.

Figure 3. Throughput with a growing number of NFs.



(a) "Double buffer" architecture.

(b) "Double buffer + semaphore" architecture.

Figure 4. Latency introduced by the framework.

application) to be allocated on a different CPU core. As a consequence, we were only able to execute 30 NFs using both the Xeon CPUs available in our server, while the remaining two logical cores were allocated to the operating system and to the vSwitch.

With this limitation, our tests confirm that FDIR could provide a considerable speedup to the system, allowing our server to reach a throughput that is up to 41% better compared to the best of the previous implementations (e.g., 6.05Gbps with 64B packets and 4 NFs). However, this gain tends to decrease when adding more NFs (particularly when we start to allocate NFs on the second CPU, which forces the traffic to traverse the QPI bus), reaching a point, with 30NFs, in which this solution does no longer provide advantages at all.

The latency measured in this test case is slightly better than the one provided in Figure 4(a), as packets coming from the NIC are immediately delivered to the NFs without passing in the vSwitch. However, for space concerns, it is not reported in the paper.

D. Isolated buffers + semaphore

Figure 3(d) shows the throughput achieved through the implementation providing isolation among NFs. Comparing this graph with that depicted in Figure 3(b), it is evi-

dent a deterioration in performance, as a consequence of the additional copies needed to guarantee traffic isolation among NFs. Instead, latency looks very similar to the one presented in Figure 4(b).

VI. Related work

NetVM [5] is a platform built on top of KVM and DPDK, designed to efficiently provide network traffic to NFs deployed as different virtual machines (VMs). The onerous constraints (in terms of hardware resources) imposed by full fledged VMs does not allow NetVM to support thousands of NFs running together on the same physical server. Moreover, the NetVM hypervisor exploits several threads to provide packets to NFs, while our vSwitch uses a single CPU core as we would like to allocate all the others to the NFs.

Also ClickOS [6] is targeted at efficiently steering traffic among many NFs deployed on the same server. In fact, the paper defines an efficient packet exchange mechanism between a vSwitch (based on VALE [7]) and NFs, which are implemented as tiny VMs based on Click [8]. Hence, NFs require limited hardware resources and bootstrap time. Unlike our work, [6] does not consider different architectures to be used according to the number and the type of NFs deployed, as well as the number of NFs supported is reduced compared to that shown in this paper.

Finally, [9] proposes to execute, in the data-plane of an edge router, fine-grained applications associated with a specific user. However it limits its scope to edge routers, while our proposal considers servers running generic NFs.

VII. Conclusion

This paper focuses on the case in which a massive number of (tiny) network function instances are executed simultaneously on the same server and presents five possible implementations, each one with specific operating characteristics, of a system that moves efficiently packets across the many NFs running on the server itself. All the proposed implementations are based, as much as possible, on the features offered by the Data Plane Development Kit, a framework recently proposed by Intel to efficiently implement data plane applications.

Results obtained, particularly in terms of throughput, are quite satisfying for almost all the implementations proposed, confirming the goodness of the primitives exported by the DPDK; only in few cases we spotted some limitations which are specific of our target domain. From the point of view of the latency, we experienced huge packet traveling times when the server was packed with many NF active at the same time. In general, when the number of NF exceeded 100, the average latency experienced by the packets may become unacceptable in real implementations.

To our view, this suggests that our particular use case, with a massive number of (tiny) NFs, may not be satisfied with the current generation of the hardware, in which CPUs are dimensioned for a few, fat, jobs, while we have here many, tiny tasks. This suggests that our future investigations should take into consideration different hardware platforms, such as the ones with a massive number of (tiny) cores, which may be more appropriate for our case. This, for instance, is one of the objectives of the UNIFY project with respect to task T5.3, focused on the feasibility analysis of different hardware solutions with respect to various types of workloads.

Acknowledgment

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.

References

[1] (2014) Intel dpdk - programmer's guide. [Online]. Available: http://dpdk.org/doc/intel/dpdk-prog-guide-1.7.0.pdf

[2] A. Császár, W. John, M. Kind, C. Meirosu, G. Pongrácz, D. Staessens, A. Takács, and F.-J. Westphal, "Unifying cloud and carrier network: Eu fp7 project unify," in *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC '13)*. IEEE Computer Society, 2013, pp. 452–457.

[3] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel." in *USENIX Annual Technical Conference*, 1996, pp. 99–112.

[4] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending networking into the virtualization layer," in *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, October 2009.

[5] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: High performance and flexible networking using virtualization on commodity platforms," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 445–458.

[6] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 459–473.

[7] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 61–72.

[8] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, ser. SOSP '99. New York, NY, USA: ACM, 1999, pp. 217–231.

[9] F. Risso and I. Cerrato, "Customizing data-plane processing in edge routers," *2012 European Workshop on Software Defined Networks*, pp. 114–120, 2012.