An Efficient Data Exchange Algorithm for Chained Network Functions

Ivano Cerrato, Guido Marchetto, Fulvio Risso, Riccardo Sisto, Matteo Virgilio
Department of Control and Computer Engineering
Politecnico di Torino, Torino (Italy)
E-mail: {ivano.cerrato, guido.marchetto, fulvio.risso, riccardo.sisto, matteo.virgilio}@polito.it

***Abstract*—In-network function chaining often involves the deployment of multiple applications into a single, possibly multi-tenant, middlebox. This approach has gained much interest since new network paradigms, such as Software Defined Networking (SDN) and Network Function Virtualization (NFV), have been proposed to virtualize resources as well as network functions. In this scenario, it is very common to move data (e.g., packets) from an application to another by means of a switching module that is in charge of chaining network functions in the correct order, also ensuring an adequate level of isolation between any two virtualized components. With this purpose in mind, this paper proposes an efficient algorithm to handle the communication between the internal soft-switch and the heterogeneous network functions that are executed on the same server. Our proposal is designed with the aim of dealing with high speed packet processing, hence an extensive performance evaluation is also provided to prove the goodness of our solution in this context.**

I. Introduction

Recently we assisted to the consolidation of two new network paradigms, namely Software Defined Networking (SDN) and Network Functions Virtualization (NFV), which assign much more importance to the role of the software in both the data and control plane of the network. Briefly [1], SDN is based on the separation between the control and the data plane of the network; the former is transformed into an open and programmable platform that allows many actors to finely control the forwarding decisions taken in any portion of their network. Instead, NFV focuses on the problem of consolidating and optimizing the processing of the network traffic that needs to traverse several middleboxes, each one implementing a specific function[1] (e.g., NAT, firewall, etc), with a huge impact in terms of costs, reliability and complexity of the network. NFV proposes to transform the network functions that today are running on proprietary equipment into a set of software images that could be installed on general purpose hardware, hence leveraging high-volume standard servers (e.g., x86-based blades) and computing/storage virtualization. This results in higher flexibility for applications, as well as in lower capital and operating costs for the hardware, since many different functions can be deployed on the same middlebox instead of being forced to exploit dedicated network middleboxes as in the past.

A direct consequence of this flexibility is that two packets may traverse two different function chains, e.g., one packet has to be handled by a WAN accelerator, while the other one, which carries HTTP traffic, has to be handled by a web cache, then both packets have to traverse a firewall. This requires the presence, within the network node, of a module that classifies the traffic and sends it to the proper functions. This is the *virtual switch* component shown in Figure 1. Furthermore, as network functions can modify the packets (e.g., an HTTP request packet is replaced by a packet that asks for that content from the nearest cache by means of a proprietary protocol), the classification must not only be done as soon as the packet enters the node, but it must be re-executed each time the traffic leaves an application. In fact, the classifier in the virtual switch module cannot know, when the packet enters the middlebox, the entire sequence of applications it has to traverse during its journey within the network node. As a consequence, each application, after having handled a packet, must send it back to the virtual switch, which can determine which is the next function that has to be traversed or, if it has already been handled by all the required applications, can send it on the network.

Figure 1 depicts the journey of a packet through a chain of data plane applications in a middlebox, and also shows that the number of functions installed within the middlebox is generally higher than those traversed by a single packet. However, all the packets traverse the virtual switch multiple times, which suggests that this module should be carefully designed in order not to become the bottleneck of the system (more insights on this choice will be detailed in Section III-A).

In this context, our contribution is to propose and evaluate an efficient way for moving data between the virtual switch and a generic network application, which is based on a single lock-free shared circular buffer. In order to achieve high performance the system is designed so as to: *(i)* exploit cache locality (both for code and data) as much as possible and *(ii)* limit the number of context switches since their cost would introduce an excessive overhead [2]. In addition, since multi-tenant network nodes are envisaged, the algorithm should take into account that function chains may involve applications in-

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.

[1]In the rest of this document, the terms *data plane application*, *application* and *function* will be used interchangeably. Instead, *network function chain* is the result of many network functions chained one after the other.

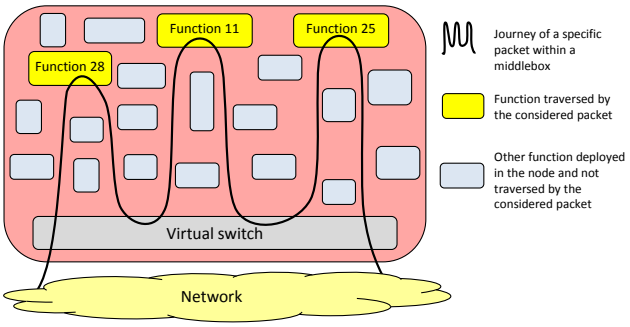


Fig. 1. Journey of a packet through a function chain deployed in a middlebox.

stalled/developed by different tenants; then, the data exchange algorithm must guarantee *traffic isolation* between functions, so that a function can only access the portion of network traffic that is expected to flow through it, hence limiting the potential hazards due to malicious applications.

The problem of envisioning an efficient data exchange mechanism between different network functions operating within the same server is one of the topics of the EU-funded FP7 project UNIFY [3], which aims at providing full network and service virtualization to enable rich and flexible services. Particularly, efficient data exchange is one of the requirements mentioned in the deliverable D5.1 [4], which provides the functional specification of the *universal node*, i.e., a platform that is potentially able to deliver both computing and network virtualized services.

The rest of the paper is organized as follows: Section II briefly recalls the most relevant work that inspired our proposal. Section III dives into the design of our proposal, presenting the core concepts of the data communication algorithm. Section IV shows some evaluation results while Section V draws conclusions.

II. Related Work

The efficient *lock-free* implementation of FIFO queues has been investigated in several research papers. However, all the solutions proposed so far are not optimized for function chains in network middleboxes because they are usually based on unidirectional flow of data according to the *producer-consumer* paradigm. For instance, [5], [6] and [7] propose lock-free algorithms that operate on FIFO queues managed as non-circular linked-lists. Similar proposals can be found in [8] and [9], which require also to manage a pool of pre-allocated memory slots. However, since in network function chains a packet always goes from the virtual switch to the network function and then back to the virtual switch, these algorithms require the applications to remove the data just received from a first queue and to write it into a second queue used for sending the data back. This implies that data is always copied once in this trip, which may limit the throughput of the system particularly when several functions have to be traversed (hence several copies have to be completed).

Another possible way to efficiently exchange data between applications can be seen in the context of a lock-free operating system, in which [10] and [11] present a single producer/consumer and a multi-producer/multi-consumer algorithm to manage circular FIFO queues. Similar proposals have been made also in [12] and [13], whose algorithms have been designed to operate in contexts where many processes can concurrently insert or remove items from a shared buffer. However, those proposals are not applicable in our case as we need to provide isolation between the network functions, which is not guaranteed by a unique shared buffer.

Aside from the pure buffering mechanism, another important aspect to consider when moving data between different processing entities is the interaction of the queuing mechanism with the rest of the system. Particularly, the algorithm should avoid an excessive overhead due to context switches or improper memory access patterns, which impacts on the effectiveness of the CPU cache. For instance, those techniques are taken into high consideration in the Intel DPDK library [12], which has been explicitly designed for data plane applications. In fact, DPDK implements effective data batching mechanisms to improve performance, as well as algorithms designed to exploit memory locality.

MCRingBuffer [14] defines an algorithm to exchange data between one producer and one consumer running on different CPU cores that is particularly efficient with respect to memory access patterns. For instance, it defines a cache-line protection mechanism that places the shared and local variables for the producer and the consumer in different cache lines. Furthermore, the processes use mainly local variables to access the buffer, and only when the buffer is *potentially* empty/full they actually read the shared variables in order to realign their local copies. Similar techniques are exploited in our algorithm as well.

Finally, works such as ClickOS [15] (based on the VALE virtual switch [16]) and Xen [17] address the problem of efficiently exchanging packets between different entities such as virtual machines (VM) running on the same physical server, which can be seen as similar to our problem of implementing network function chains. However, their intrinsic architecture is designed for packet destined to or generated from the VMs, without forcing the traffic to return to the virtual switch. This implies different architectural choices such as different buffers for packets in different directions, albeit integrated with sophisticated data exchange mechanisms (e.g., in [17]) based on exchanging memory pages rather than copying the packet between hypervisor and the VM.

As a final remark, it is worth pointing out that this paper focuses on the problem of efficiently moving packets between different functions within a network middlebox, while it does not consider the problem of efficiently receiving/sending packets from/to the network. This aspect, which is orthogonal to our proposal, is instead considered in [18] and [19].

III. Algorithm

This section provides an overview of the proposed algorithm, introducing first the goals and constraints that are derived from the use case presented in Section I, followed by a detailed description of the algorithm itself.

A. Design choices and architecture

As stated in Section I, flexible function chains require a fast and efficient communication mechanism to move traffic between the virtual switch and the network functions (and then back), which translates into the necessity of a dedicated data dispatching mechanism, being this component one of those that most influence the performance of the system.

The fundamental choice for this mechanism is between a distributed architecture, which looks more appropriate for a component that may become the bottleneck of the system, or a more traditional (and centralized) architecture. For instance, while the centralized architecture may be translated into a virtual switch that has to dispatch packets to the various applications and receive them back in order to determine the next processing step, the distributed architecture allows each function to determine autonomously which is the function that follows and, consequently, it is able to send the traffic directly to it, in a completely distributed fashion.

Although more appealing, the distributed architecture has several problems. First, the necessity to synchronize all the classifier instances deployed in each function when some chains change (e.g, new functions are added/removed, new flows are added that require new service paths, etc.). Second, the necessity to arbitrate the transfer of packets between each function and the next one, as each function may receive input traffic from multiple sources. While the above problems could lead to a noticeable number of implementation issues (with an expected impact in terms of performance) for a possible distributed architecture, the necessity to *isolate* a function from the following ones (Section I) puts definitely to an end the distributed architecture. In fact, a distributed architecture would not be able to prevent a network function F_1 from accessing the packets directed to function F_2, with potential security risks as an application may be able to modify traffic that is not under its responsibility. This issue can be solved by means of an intermediate entity that is in charge of isolating network functions from one another, giving each application only the visibility on its own traffic. This, in turn, makes the centralized architecture, based on the virtual switch, the most appropriate.

Our data exchange mechanism is based on a set of lock-free ring buffers, each one shared by the virtual switch (which is named *Master* in our algorithm) and a single function instance (named *Worker*), as shown in Figure 2. Each shared buffer is used for the communication in both directions: since data provided by the Master to a Worker will eventually come back to the Master itself, the algorithm allows the Worker to return those data back without any copy. Instead, when data received from a Worker must be sent to another Worker of the chain, the Master will make a copy of the data from the buffer shared with the first Worker to the buffer shared with the second one, as each Worker has access only to its own buffer. This design enables higher throughput for the function chain thanks to the capability to limit the number of data copies in the system, which, for each packet, are equal to the number of functions

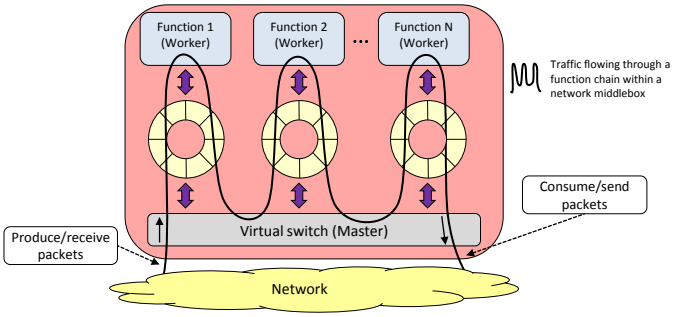


Fig. 2. Deployment of the algorithm within a middlebox.

N traversed by the packet itself minus one[2]. It is worth noting that a solution with a single buffer shared among the Master and all the Workers of the chain would allow the packet to be moved with zero-copy from Worker to Worker, but it would not ensure an adequate level of isolation: in fact, a Worker could access data that should not flow through it.

To better understand the overall functioning of the system, let us describe the scenario depicted in Figure 2, where packets have to flow through a number of Workers, each one executing a different kind of network application. When a packet enters the node, the Master determines the first application that must process the data based on one or more field of the packet itself. Once the packet has been inserted into the proper buffer, the corresponding Worker will process it by applying its logic, possibly altering the content of the packet. After marking the data as "consumed", the Master will be able again to handle the packet and decide which is the next application. At the end of the chain, when all the Workers have successfully completed their task, the packet is finally forwarded by the Master on the network.

Moving to the execution model, the Master operates in polling mode, i.e. it continuously reads a new packet and copies it into the buffer shared with the target Worker. This mode is appropriate in our working conditions, where the network node is supposed to process an huge amount of traffic. Instead, with a blocking model the Master would need to be woken up as soon as new data arrive, which requires interrupt-like mechanisms to start the processing. This would be too penalizing because of the excessive overhead of those inter-process communications (e.g., interrupts) and the associated context switches. Vice versa, the traffic entering a specific Worker is potentially a small portion (e.g., a single tenant) compared to the one handled by the Master, then in this module a blocking model looks appropriate. In particular, we have a semaphore shared between the Master and the Worker, used by the first module to wake up the application at the "right time" (more details in the next section). As a consequence a Worker will suspend itself when it has no more data to be processed, until the Master wakes it up again; this way, CPU resources can be used by Workers actually having data to be processed.

[2]In real systems one additional copy is required, because the packet has to be copied from the memory of the network interface card to the shared buffer of the first Worker, bringing the total number of copies to N.

B. Algorithm description

The proposed data exchange algorithm is based on the above mentioned principles and it requires the Master and each Worker to share some indexes and a semaphore, in addition to the circular buffer itself. Its main operation phases are summarized in Algorithm 1 (with respect to the Master portion) and Algorithm 2 (for the Worker portion), which suppose the presence of a Master and a single Worker, while its extension to the multi-worker case is trivial. The shared buffer is made of M slots, whose size is equal to the maximum packet size of the system; shorter packets are copied aligned at the beginning of the slot while the remaining bytes are left unused.

As evident from Algorithm 1, the Master cyclically repeats the following three main operations: *(i)* produces new data[3] and immediately dispatches it to the Worker through the shared buffer (lines 9-10), *(ii)* reads from the buffer the data already processed by the Worker (line 12), and finally *(iii)* wakes up the Worker if it has been sleeping for too long and there are data to be processed (line 13), in order to avoid data starvation and packet aging, as better detailed below. It is worth noting that the Master produces a certain number of consecutive data in order to better exploit the cache locality (the *while* loop at line 5). Furthermore, if the buffer is full, it stops producing new data (lines 6-7) and immediately starts to remove from the buffer those data that the Worker has already handled.

Algorithm 1 Executing the Master

```
 1: Procedure master.do()
 2:
 3: while true do
 4:    i = 0
 5:    while i < N  do
 6:       if (buffer.isFull()) then
 7:          break
 8:       end if
 9:       data ← master.produceData()
10:       master.writeDataIntoBuffer(data)
11:    end while
12:    master.readDataFromBuffer()
13:    master.checkForOldData()
14: end while
```

In addition to the high-level overview of the algorithm depicted in the pseudo-code, some further details have to be considered to better clarify its operation. First, we must avoid that a Worker is woken up for each single packet that needs to be processed. For instance, it would be more convenient to queue several packets in the Worker's buffer before waking it up, which would allow batch processing in the Worker (hence achieving better efficiency because of code and data locality) and would limit the number of context switches in the system (when a Worker is suspended because of the lack of data and another is scheduled for execution). In this respect, our algorithm defines a *Master threshold* which represents the minimum number of packets that need to be waiting for service in the buffer before waking the Worker up. Second, we must avoid that the packets of a Worker with a limited amount of traffic would never be serviced because the Master threshold is never reached. To comply with this requirement, the algorithm stores the timestamp of the oldest packet present in the shared buffer and checks if the buffer holds packets that appear too old: in this case the Worker is woken up anyway, irrespective of the Master threshold.

Those conditions are checked in the functions `writeDataIntoBuffer()` and `checkForOldData()`. Particularly, the former does the following: (*i*) if the buffer is empty, stores the current time in a variable; (*ii*) copies the new data in the first free slot (which is pointed by a shared index); (*iii*) if the Master threshold has been reached and the Worker is not already up and processing data, wakes it up. Instead, the `checkForOldData()` function is in charge of checking for the presence of too old packets waiting to be served, which requires to check if (*i*) the buffer contains one or more data, (*ii*) the Worker is not already running, (*iii*) the timestamp associated with the shared buffer has exceeded a predefined threshold.

Both functions need to know whether the Worker is still suspended or it is already running. This is done through a variable shared between the Master and the Worker that tracks the status of the latter, named `workerStatus` in the pseudo-code of Algorithm 2. This variable is *set* by the Master just before signaling the Worker to wake up, while the Worker *resets* the variable just before going to sleep. In this way, the Master can test this shared variable to have an indication about the status of the Worker and then wake it up only when necessary.

Algorithm 2 Executing the Worker

```
 1: Procedure worker.do()
 2:
 3: while true do
 4:    waitForWakeUp()
 5:    processed_packets ← 0
 6:    while isTherePacket() do
 7:       if (processed_packets ≥ WORKER_PKT_THRESHOLD)
          then
 8:          processed_packets ← 0
 9:          updateSharedIndex()
10:       end if
11:       buffer.process()
12:       processed_packets++
13:    end while
14:    updateSharedIndex()
15:    workerStatus ← WAIT_FOR_SIGNAL
16: end while
```

Algorithm 2 details the operations of the Worker. In particular, when it wakes up, it processes the data into the buffer until data are available (lines 6-13). When it finishes (line 14), or when it has already processed at least a given amount of data (line 9), the Worker updates a shared index, so that the Master can consume the data just processed by the Worker itself. This way, also the Master is able to implement batched reads, i.e., consuming several packets from the shared buffer

[3]It is worth noting that *producing new data* corresponds, in fact, to reading packet from the network interface card.

at once, in order to better exploit data and code locality and improve efficiency.

Notice that this batching mechanism is implemented in a different way compared to the one that refers to the data sent by the Master to the Worker. In fact, while in previous case the Worker is woken up when the amount of data into the buffer is higher than the *Master threshold*, in this case we update a shared variable (i.e., the index used by the Master to know the amount of data ready to be consumed in the buffer) only periodically, instead of incrementing its value by one each time the Worker processes a packet.

IV. Experimental results

This section reports on the results of several tests aimed at identifying the maximum throughput that can be achieved by the proposed algorithm in several test conditions. These tests were executed on a workstation with 16 GiB of memory, CPU Intel i7-3770 @ 3.40 GHz (four physical cores plus hyperthreading) and OS Ubuntu 12.10, kernel 3.5.0-17-generic, 64 bits. In all the tests an entire core was dedicated to the master, which represents the most critical component of the system because it has to dispatch packets to all the Workers, depending on the function chain experienced by each packet. Instead, for what concerns the Workers, they were distributed among the cores (except that running the Master) in a fashion that maximises the throughput of the system.

Each test lasted 100 seconds and was repeated 10 times. The results are averaged and reported in the graphs shown in the following. Each graph is provided with a bars view and a points-based representation of the maximum throughput. The first representation is referred to the left y axis, which reports the throughput in millions of packets per second, while the second one is referred to the right y axis, where the throughput is measured in Gigabit per seconds. The data exchanged between the Master and the Workers consists of network packets of different sizes. Moreover, the chain of Workers traversed by each packet is statically defined and the tests are repeated with different the lengths of the chain.

Figure 3 provides the throughput in four different test conditions, considering function chains with a different number of Workers and forcing the packet to traverse them all. This provides an insight of the forwarding capabilities of a network node in case long function chains are configured. As expected, the throughput decreases when increasing of the number of Workers, which originates from the increased amount of time spent by the Master to move packets from each Worker to the following one. Furthermore, the presence of many Workers has also an impact on data locality, as more buffers are allocated and consequently there is a higher probability for the CPU to experience cache misses, even if our algorithm tends to limit their impact through batch processing.

Particularly, Figure 3(a) shows the throughput that could be achieved in ideal scenarios, that is: with Workers that do not actually access the packet[4], referred to as "dummy" Workers in the following, and with a single packet in memory. The latter point means that the Master reads always the same packet in memory and copies it into the buffer of the first Worker, thus reducing the impact of the CPU cache miss experienced at the beginning of the chain. This provides an ideal view of the system compared to the actual scenario of a network middlebox, were distinct packets are received from the network.

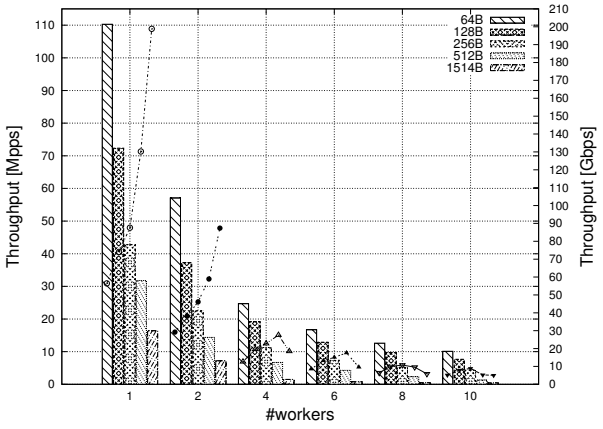
Starting from this ideal situation, the following experiments move into more realistic scenarios. In particular, Figure 3(b) refers to Workers that access the packet content and calculate a signature across the first 64 bytes of the packets, hence emulating the operation that most network applications perform on the first bytes (i.e., the headers) of each packet. The main reason of the declining throughput of Figure 3(b) (compared to the ideal case of Figure 3(a)) is the increased number of cache misses in the L1 and L2 caches of the CPU core where the Worker is running. In other words, the additional load in the Worker due to the computation of the signature seems to have a marginal impact compared to the time needed by the CPU to move data from the cache of the core where the Master is running to the core of the Worker.

Figure 3(c) refers to a scenario with dummy Workers (such as in case (a)), but the Master reads data to be injected into the chain from a buffer containing 1 million of packets. Also in this case the reduced throughput compared to the ideal case is due to memory access patterns, as the Master will very likely experience frequent cache misses when reading packets at the beginning of the chain. The obtained results confirm that this modification alone can halve the throughput of our system, particularly when the packet has to traverse a limited number of Workers, while in case of longer chains this additional overhead at the beginning is amortized by the cost of the rest of the chain.

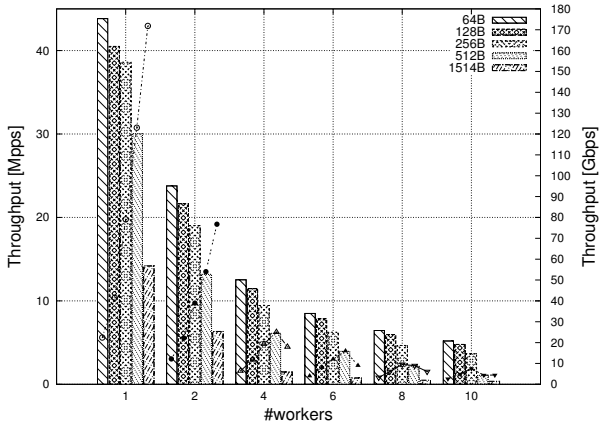
Finally, results in Figure 3(d) refer to an even more critical scenario, where the Master reads packets from a buffer storing 1 million of packets and the Workers are the above described realistic ones considered for Figure 3(b). Notice how also the performance obtained in this case is very satisfying, reaching about 38 millions of packets per second with 64 bytes packets when a single worker has to be traversed. In addition to this, we have to consider that the Master uses a single CPU core and we do not want to exploit, by design, more CPU cores as we would like to allocate them to the Workers, which will host the network functions.

Figure 4 shows, in the same conditions of the results provided in Figure 3(d), the internal throughput of the chain, namely the total number of packets per second moved by the Master, with an increasing number of Workers. This picture gives an insight of the processing capabilities of the Master that slightly increase with the number of workers, thus proving the goodness of our algorithm as the number of packets that our algorithm successfully processes does not depend on the number of Workers. In fact, a packet that travels across multiple Workers is likely to experience cache misses at the beginning of the chain, while it may be found in cache in the following processing steps. Therefore, an higher number of Workers mitigates the cache miss problem, hence allowing to achieve higher throughput.

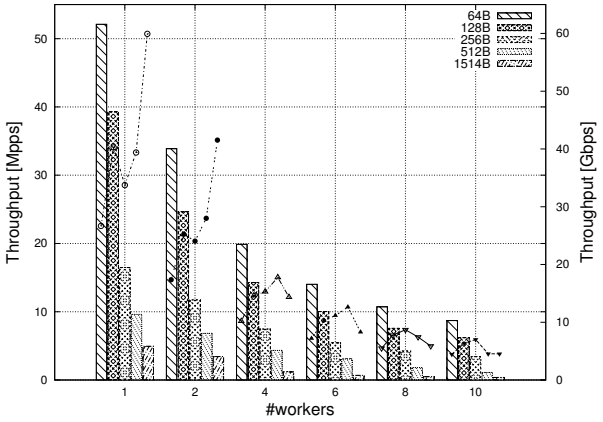
[4]It is worth remembering that the algorithm allows a Worker to send back packets to the Master without actually accessing the packet content.



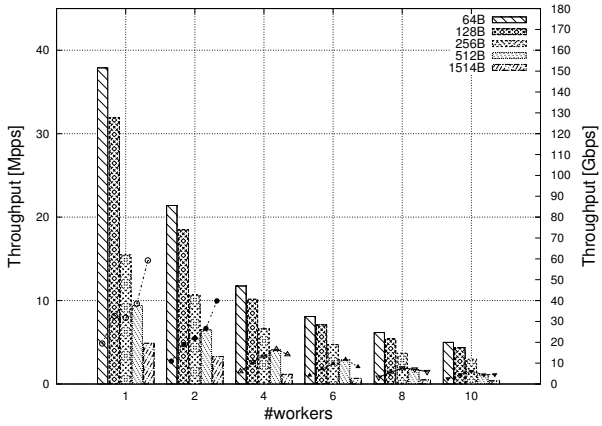
(a) Dummy Workers and a single packet in memory.



(b) Real Workers and a single packet in memory.



(c) Dummy Workers and 1M packets in memory.



(d) Real Workers and 1M packets in memory.

Fig. 3. Throughput of the function chain.

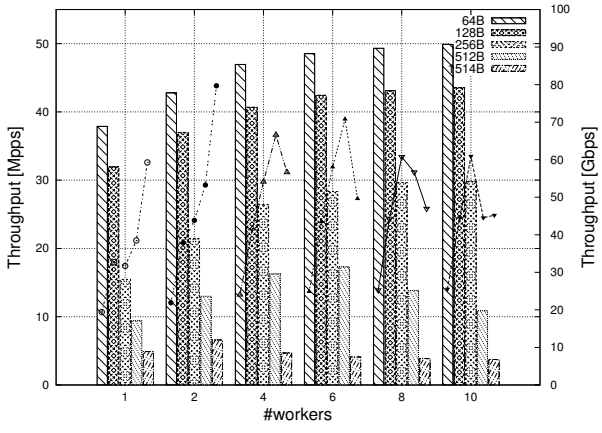


Fig. 4. Internal throughput of the function chain.

The last test, shown in Figure 5, refers to the throughput obtained when dispatching packets (of different sizes) among a growing number of *chains*. This test emulates the fact that different packets may travel across a different number of functions in the server, as we expect that their total number will be much higher than the number of functions traversed by each packet. This is different from the previous case in which packet had to traverse a growing number of functions, but the function chain is fixed. This test is particularly critical with respect to the memory access patterns, thus stressing the CPU cache, because (*i*) the Master has to read packets from an high number of buffers and (*ii*) the packets read by the Master are likely to be copied in different buffers for the next processing step.

In this test, packets are provided in a round robin fashion to a growing number of function chains, each one composed of four cascading Workers. Chains have been randomly generated, but on average 33% of the Workers are shared with other chains, which means that (in average) the traffic of three chains exploit

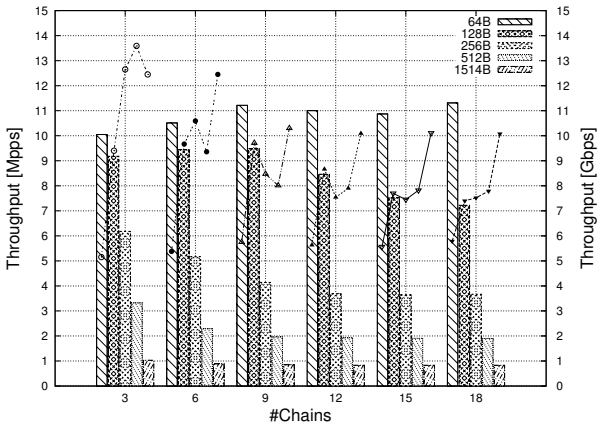


Fig. 5. Throughput with a growing number of chains.

eight Workers, bringing the total number of Workers active in our test to 48 in the most complex case (18 chains). Workers have been allocated across six cores of the CPU, in a way that minimizes the number of times a packet has to be copied from a core to another in order to limit CPU cache synchronization operations among cores. Figure 5 shows that, as expected, the final throughput tend to slightly decrease with the number of chains, except for the case of 64 bytes packets in which the throughput increases. Some more investigations are under way to explain the throughput increase in case of short packets.

Finally we compare the advantages, in terms of performance, of our shared buffer against a traditional mechanism based on two unidirectional buffers between Master and Workers, while still maintaining the other design goals (e.g., batch processing, etc.) listed in Section III. Our results confirm that the performance degradation due to the additional data copy (packets have to be copied by the Worker from the first buffer to the second one) has a noticeable impact on the overall performance. In fact, the throughput of the entire chain halves in the ideal case (dummy Workers, a single packet in memory) as shown by comparing Figure 3(a) to Figure 6(a), while drops of about 30% in case of real conditions (real workers, 1M packet buffer) as shown in Figure 3(d) vs Figure 6(b). This confirms the advantages of our algorithm at least in our application scenario.

V. Conclusion

A novel algorithm has been described to efficiently move data between network virtual function implementations (the Workers) by means of a switching module (the Master). The algorithm provides both performance to the whole system and traffic isolation among the different Workers. One of the peculiarities of this approach is that data are sent to a Worker and then returned back to the Master for further processing with zero-copy, by means of a single lock-free buffer. A form of batching has also been introduced in order to amortize the cost of context switches, while a safeguard mechanism avoids packet starvation in case of Workers traversed by a limited amount of traffic. Our algorithm has been evaluated with a wide range of experiments on a prototype designed to characterize its behavior, with promising results. As a future activity, we are planning to implement the algorithm in an existing softswitch that handles function chaining and network functions virtualization in a real cloud environment, which will offer the possibility to validate the algorithm within the framework provided by the UNIFY project.

References

[1] F. Risso, A. Manzalini, and M. Nemirovsky, "Some controversial opinions on software-defined data plane services," in *Proceedings of the First Workshop on Software Defined Networks for Future Network Services (SDN4FNS)*, ser. SDN4FNS13. IEEE, November 2013, pp. 1–7. [Online]. Available: http://dx.doi.org/10.1109/SDN4FNS.2013.6702558

[2] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proceedings of the 2007 workshop on Experimental computer science*, ser. ExpCS '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1281700.1281702

[3] 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*. IEEE Computer Society, 2013, pp. 452–457. [Online]. Available: http://dx.doi.org/10.1109/UCC.2013.89

[4] The UNIFY Project, "Deliverable d5.1: Universal node functional specification and use case requirements on data plane," http://www.fp7-unify.eu/index.php/results.html, March 2014.

[5] Stone, "A simple and correct shared-queue algorithm using compare-and-swap," *SC Conference*, vol. 0, pp. 495–504, 1990.

[6] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '96. New York, NY, USA: ACM, 1996, pp. 267–275. [Online]. Available: http://doi.acm.org/10.1145/248052.248106

[7] A. Gidenstam, H. Sundell, and P. Tsigas, "Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency," in *Proceedings of the 14th international conference on Principles of distributed systems*, ser. OPODIS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 302–317. [Online]. Available: http://dl.acm.org/citation.cfm?id=1940234.1940266

[8] S. Prakash, Y. H. Lee, and T. Johnson, "A nonblocking algorithm for shared queues using compare-and-swap," *IEEE Trans. Comput.*, vol. 43, no. 5, pp. 548–559, May 1994. [Online]. Available: http://dx.doi.org/10.1109/12.280802

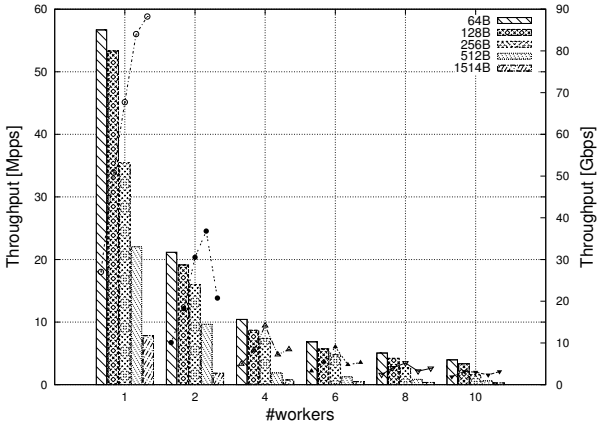
[9] M. Hoffman, O. Shalev, and N. Shavit, "The baskets queue." in *OPODIS*, ser. Lecture Notes in Computer Science, E. Tovar, P. Tsigas, and H. Fouchal, Eds., vol. 4878. Springer, 2007, pp. 401–414. [Online]. Available: http://dblp.uni-trier.de/db/conf/opodis/opodis2007.html

[10] H. Massalin and C. Pu, "Threads and input/output in the synthesis kernel," in *Proceedings of the twelfth ACM symposium on Operating systems principles*, ser. SOSP '89. New York, NY, USA: ACM, 1989, pp. 191–201. [Online]. Available: http://doi.acm.org/10.1145/74850.74869

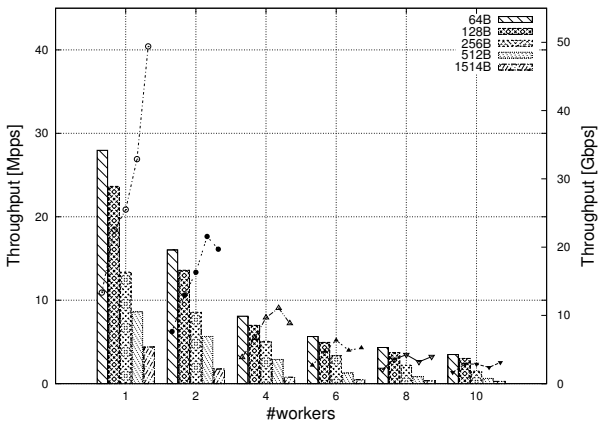
[11] ——, "A lock-free multiprocessor os kernel," *SIGOPS Oper. Syst. Rev.*, vol. 26, no. 2, pp. 108–, Apr. 1992. [Online]. Available: http://dl.acm.org/citation.cfm?id=142111.964561

[12] Intel. (2012) Data plane developer kit - programmers guide. [Online]. Available: http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/intel-dpdk-programmers-guide.html

[13] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," in *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '01. New York, NY, USA: ACM, 2001, pp. 134–143. [Online]. Available: http://doi.acm.org/10.1145/378580.378611



(a) Dummy Workers and a single packet in memory.



(b) Real Workers and 1M packets in memory.

Fig. 6. Throughput of the function chain when unidirectional buffers are used.

[14] P. P. C. Lee, T. Bu, and G. P. Chandranmenon, "A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring," in *IPDPS*, 2010, pp. 1–12.

[15] J. Martins, M. Ahmed, C. Raiciu, and F. Huici, "Enabling fast, dynamic network processing with clickos," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 67–72. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491195

[16] 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. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413185

[17] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945462

[18] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi-core systems," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 218–224. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879169

[19] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342830