An efficient data exchange mechanism for chained network functions

An Efficient Data Exchange Mechanism for Chained Network Functions

Ivano Cerrato[a], Guido Marchetto[a,*], Fulvio Risso[a], Riccardo Sisto[a], Matteo Virgilio[a], Roberto Bonafiglia[a]

[a] *Department of Control and Computer Engineering*
Politecnico di Torino
Torino, Italy, 10129

Abstract

Thanks to the increasing success of virtualization technologies and processing capabilities of computing devices, the deployment of virtual network functions is evolving towards a unified approach aiming at concentrating a huge amount of such functions within a limited number of commodity servers. To keep pace with this trend, a key issue to address is the definition of a secure and efficient way to move data between the different virtualized environments hosting the functions and a centralized component that builds the function chains within a single server. This paper proposes an efficient algorithm that realizes this vision and that, by exploiting the peculiarities of this application domain, is more efficient than classical solutions. The algorithm that manages the data exchanges is validated by performing a formal verification of its main safety and security properties, and an extensive functional and performance evaluation is presented.

Keywords: parallel algorithms, high speed packet processing, data exchange mechanism, network function virtualization

1. Introduction

New paradigms have recently emerged that aim at transforming the network into a more flexible and programmable platform. In particular, Network Function Virtualization (NFV) [1] proposes to replace dedicated middleboxes, used to deliver a multitude of network services by means of a growing number of (cascading) dedicated appliances, with software images that run on general-purpose servers. This allows leveraging high-volume standard machines (e.g., Intel-based blades) and computing virtualization to consolidate and optimize the processing in the data plane of the network. This results in a more flexible deployment

*Corresponding author
Email addresses: ivano.cerrato@polito.it (Ivano Cerrato), guido.marchetto@polito.it (Guido Marchetto), fulvio.risso@polito.it (Fulvio Risso), riccardo.sisto@polito.it (Riccardo Sisto), matteo.virgilio@polito.it (Matteo Virgilio), roberto.bonafiglia@polito.it (Roberto Bonafiglia)

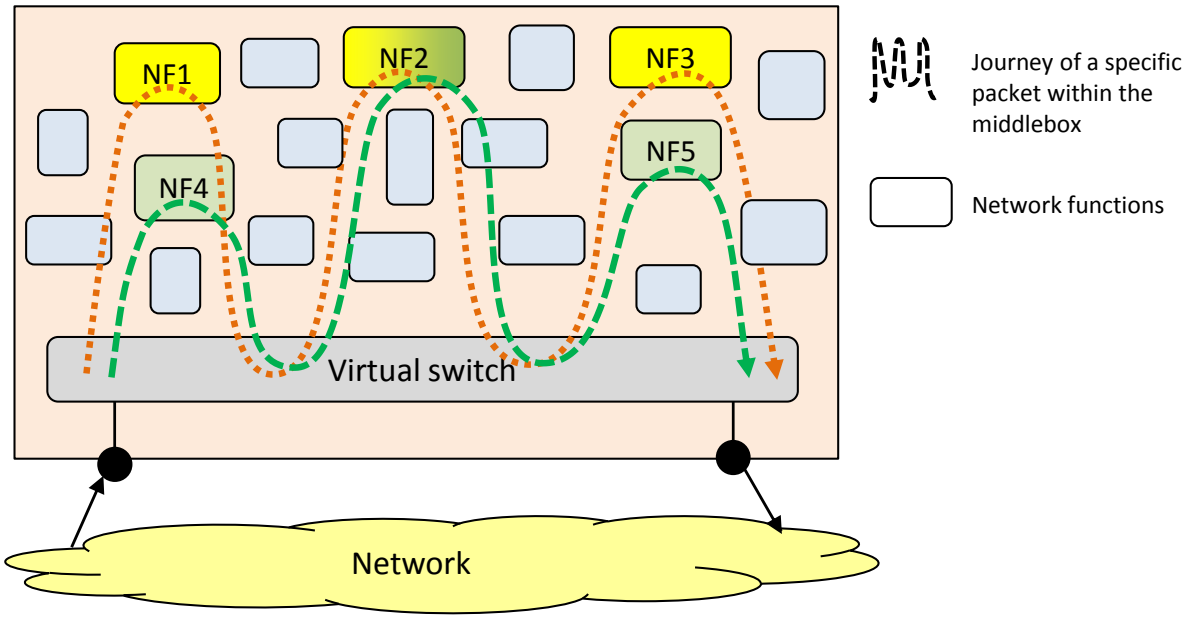


Figure 1: Function chains deployed in a middlebox.

of network applications (e.g., NAT, firewall), in lower capital and operating costs for the hardware thanks to the possibility to deploy many different (small) Virtual Network Functions (VNF) on the same (standard) computer, and in simpler and more reliable networks. In addition, while appliances are often dedicated to a single tenant, servers can be multitenant, hence being able to host services belonging to different actors [2] (e.g., a traffic monitor belonging to the operator and a firewall belonging to a given company). This brings even more advantages in terms of consolidation.

When several VNFs are executed in the same server, an incoming packet can traverse an arbitrary number of VNFs before leaving the middlebox (i.e., a *function chain*, as shown in Figure 1). The exact sequence of functions traversed by a packet can be determined only at run-time, by inspecting the packet. In fact, *(i)* packets belonging to different tenants can traverse different functions, and *(ii)* packets belonging to the same tenant can experience different paths (e.g., when only a portion of traffic needs to undergo a deep packet inspection). Packets can also be modified in transit (e.g., a NAT changes the source IP address), hence requiring that a packet is re-analyzed when it leaves a VNF, to determine what is next. As depicted in Figure 1, this requires that each server includes a module (usually referred as *virtual switch* or *vSwitch*) that classifies each packet to determine which is the next function to traverse, and then delivers the packet to it.

The several software (and virtualization) layers that packets may need to traverse within an NFV middle-

box may cause a drop of performance of virtualized functions chains, especially when several functions have to process the packets. This results in a worsening of the quality of experience for the end users, whose traffic is processed by VNFs before reaching its final destinations. Then, based on some preliminary work [3], this paper proposes and evaluates an efficient algorithm for moving network packets between VNFs consolidated on the same server and the vSwitch. Particularly, this work exploits circular lock-free First-In-First-Out (FIFO) buffers managed by ad-hoc algorithms.

Existing solutions adopted to move packets between VNFs and the vSwitch are usually based on the producer-consumer paradigm. However, since in NFV it is likely that a packet goes from the vSwitch to the VNF and then back to the vSwitch, those approaches require the VNF to copy (almost) each packet from a first receiving queue into a second queue used for sending it back. Instead, our proposal has the peculiarity of allowing VNFs to return back packets without any (expensive) packet copy, with consequent performance improvements.

Particularly, the proposed mechanism is designed to satisfy the following principles. First, guarantee *traffic isolation* between functions, so that a function can only access the portion of traffic that is expected to flow through it. This limits the potential hazards due to malicious applications and provide an effective support to multitenancy. Second, provide excellent *scalability* by allowing to consolidate a huge number of VNFs on the same server. Third, achieve high *performance* in terms of data movement speed among different VNFs, similarly to what is required in physical servers among different hardware modules [4]. Scalability and performance are obtained also by taking care of implementation details such as exploiting cache locality as much as possible and limiting the number of context switches. The correctness of the data exchange algorithm (e.g. absence of concurrency hazards) is guaranteed by means of formal verification.

The work of this paper targets VNFs that *(i)* implement a pass-through behavior (each packet received is sent again to the vSwitch), *(ii)* may drop packets, *(iii)* may generate new packets as a consequence of a packet just received (e.g., an ARP reply as a consequence of an ARP request). This allows us to cover the vast majority of network middleboxes, including for example NATs, firewalls, traffic monitors, and intrusion detection systems. Instead, applications that may need to generate new packets asynchronously, e.g., in order to open a connection with some remote service or to retransmit a TCP packet, are out of the scope of this paper and left as a future work. Note that the class of applications not supported by our proposal also includes cloud-like services (e.g., web servers, databases), since they typically need to generate new packets asynchronously and do not implement a pass-through behavior with respect to network traffic (they are in fact the final destination of network packets). Moreover, the bottleneck of these applications is typically in

the user code instead of in the network layer, hence they are not the targeted applications in this paper.

The proposed algorithm is agnostic both to the data actually exchanged and to the actors that operate on such data. Then, although the use case that motivated our research is the efficient implementation of virtualized functions chains, the algorithm can be exploited in other contexts as well, provided that modules exchanging data implement a pass-through behavior and do not send new data asynchronously.

The rest of the paper is organized as follows. Section 2 explores related existing solutions able to exchange data among different software components. Given the nature of our solution, we are particularly interested in covering FIFO queue designs and producer/consumer paradigms, in order to emphasize the differences between our work and the existing mechanisms. Section 3 details the operating principles of the proposed algorithm and the way the communication is managed by the different modules. Section 4 presents a formal verification of the data exchange algorithm, which rigorously proves its correctness from a safety and security perspective. Section 5 proposes some implementation guidelines that can be used to further improve the performance of the data exchange. Section 6 presents a wide range of experiments that validate our algorithm both in ideal conditions and in real scenarios. Finally, Section 7 concludes the paper.

2. Related Work

The efficient *lock-free* implementation of FIFO queues has been largely investigated in the past. For instance, [5] and [6] propose lock-free algorithms that operate on FIFO queues managed as non-circular linked-lists. Similar proposals can be found in [7] and [8], which also require to manage a pool of pre-allocated memory slots. However, all the solutions proposed so far are usually based on uni-directional flows of data according to the *producer-consumer* paradigm, which is not an optimal solution for managing the bi-directional data flows occurring in the virtualized environments we are considering. In fact, in these environments, a packet typically goes from the virtual switch to the VNF and then back to the virtual switch. Using classical uni-directional producer-consumer solutions requires the VNF to remove data just received from a first queue and to write them into a second queue used for sending the data back. This implies that data are 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 carried out).

Another possible way to efficiently exchange data between applications can be seen in the context of a lock-free operating system, where [9] and [10] present a single producer/consumer and a multi-producer/multi-consumer algorithm to manage circular FIFO queues. A similar idea has been proposed by Intel in the DPDK library [11] and in [12], whose algorithms have been designed to operate in contexts where many processes

can concurrently insert items into a *shared* buffer or remove them. However, those proposals are not applicable in our case because they cannot guarantee isolation between VNFs due to the presence of a unique shared buffer. Similar considerations can be made for ClickOS [13, 14] (based on the VALE virtual switch [15]) and NetVM [16], which instead targets network function chains. ClickOS uses two unidirectional queues with the necessity to copy packets once; NetVM uses two unidirectional queues between "untrusted" functions, while switching to a unique shared buffer (handled in zero-copy) among "trusted" functions, hence impairing traffic isolation requirement. MCRingBuffer [17] defines instead an algorithm to exchange data between one producer and one consumer running on different CPU cores, which is particularly interesting for its efficient implementation of memory access patterns. In fact, part of those techniques were reused in our implementation as well (Section 5).

Solutions such as Xen [18] and Hyper-Switch [19] address the problem of efficiently exchanging packets between different entities such as virtual machines (VM) running on the same server. However, their architecture is designed for packets that originate or terminate their journey in a VM, not for pass-through functions. This implies different architectural choices such as different buffers for packets flowing in different directions, albeit integrated with a complex data exchange mechanism based on swapping memory pages [18]. It is also worth mentioning that network-aware CPU management techniques have been proposed in the context of Xen for improving the performance of virtual servers hosting these network applications [20].

Virtual switches such as OpenvSwitch (OVS) [21] and the eXtensible Datapath daemon (xDPd) [22] are used to implement network function chains (as shown respectively in [23, 24]), although they appear in some way orthogonal to our proposal. In fact, they implement the classification and forwarding mechanism (either based on the traditional L2 forwarding or on the more powerful Openflow protocol [25]), but do not focus on the data exchange mechanism, which is often based on bi-directional FIFO queues (in some case a shared memory can be configured). In this respect, our mechanism can be built on top of those existing solutions to improve their data transfer capabilities, hence the overall performance of the system.

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, orthogonal to our proposal, is instead considered in [26], [27] and [11].

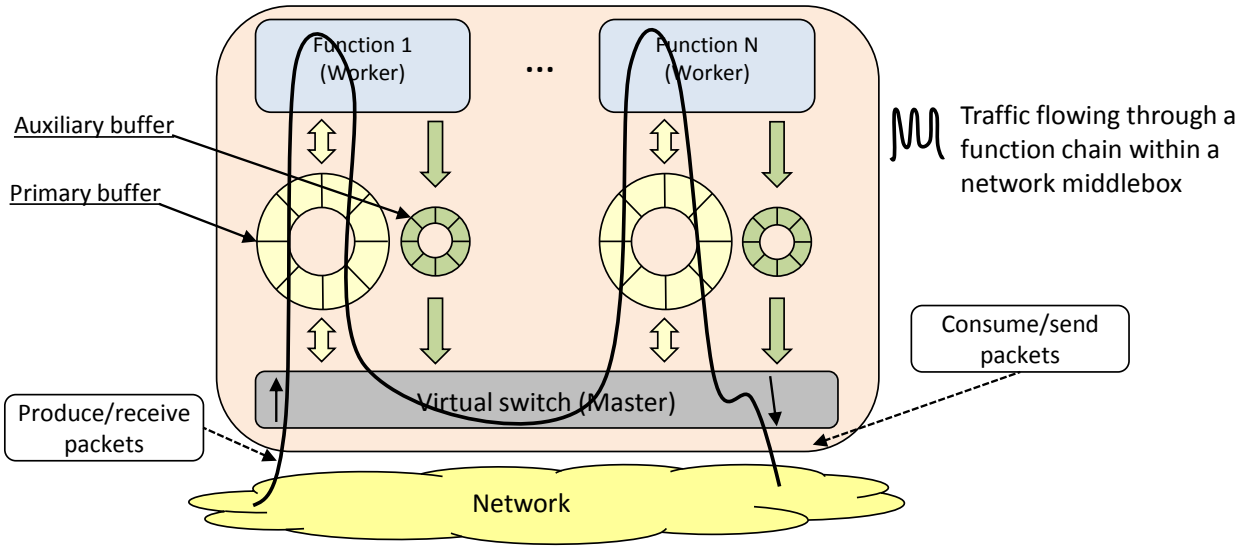


Figure 2: Deployment of the algorithm within a middlebox.

3. The data exchange algorithm

As described in Section 1, the implementation of efficient virtual functions chains within a single middlebox requires a fast and efficient mechanism to move data between the vSwitch and the VNFs. This translates into the necessity of a dedicated data dispatching algorithm, being this component one of those that has the biggest impact on the system performance.

The algorithm we designed to satisfy the above necessity is described in the remainder of this section. In particular, we first provide an overview of such an algorithm and the intuitions that are behind our proposal, then we will dive in the details of the algorithm itself.

In the context of our algorithm, we define the *Master* as the module that plays the role of the vSwitch, while VNFs are represented by modules called *Workers*. Moreover, a *token* is a generic data unit exchanged between the Master and the Workers. In fact, as stated in Section 1, although the token represents a packet in the NFV use case that motivated our research, the proposed algorithm can be actually used to exchange any kind of data, according to the specific use case implemented.

3.1. Algorithm overview

As shown in Figure 2, the proposed data exchange algorithm is based on a set of lock-free ring buffers. In particular, the Master shares two different buffers with each Worker, which are managed through different (but not independent) parts of the same exchange algorithm.

The intuition behind our proposal, which derives from the NFV use case, is the following. According to Figure 1, VNFs are pieces of software operating on the data plane of the network and that mainly process pass-through packets. In fact, VNFs receive packets from the vSwitch and, in the vast majority of cases, forward them back to the vSwitch itself with minimal (or no) changes, allowing packets to continue their journey towards the final destination. To efficiently support pass-through data, we then defined the *primary buffer* shown in Figure 2, which has the peculiarity of allowing tokens to be moved both from the Master to the Worker, and then back from the Worker to Master, without requiring any (expensive) copy of data in the Worker itself. Avoiding to copy each packet in each traversed VNF can save in fact many CPU cycles and consequently improve the performance of virtualized functions chains. Notably, in addition to VNFs operating on pass-through data, the primary buffer also supports functions that need to drop packets (e.g., firewall). Particularly, dropped packets should not be sent back to the vSwitch after their processing in the VNF.

Some network functions may need to send new packets as a consequence of a previously received packet. For example, a bridging module may need to duplicate a broadcast packet several times (e.g., once for each interface of the middlebox) and then provide all these copies to the next function in the chain. Similarly, another function may extend a packet (e.g., by adding a new header) so that it exceeds the MTU of the network; this packet must then be fragmented, and all the fragments must be sent out. Since network applications forward most of the traffic without generating new packets, we decided to keep the primary buffer as simple as possible for the sake of speed. We then defined a new second lock-free ring buffer, i.e., the *auxiliary buffer* of Figure 2, to support Workers that can possibly generate new tokens as a consequence of the data received from the Master. It is worth noting that this second buffer is unidirectional and it is only used by the Worker to provide "new" data to the Master.

Since VNFs may belong to different tenants, it is necessary to guarantee that a network function only accesses to the proper portion of network traffic. We then use a different pair of buffers per Worker in order to guarantee traffic isolation among them, as this ensures that a Worker can only access packets that are expected to flow through the Worker itself.

Each buffer slot (both primary and auxiliary) includes some flags in addition to the real data, which are used to identify the content of the slot itself; more details will be presented in the next sections. Finally, buffer slots are currently of fixed length and equal to the network MTU size; however, the extension of the algorithm to handle variable slot sizes, tailored to the length of the packet actually received, is trivial.

3.2. Execution model

The Master operates in polling mode, i.e., it continuously checks for new tokens and inserts them into the primary buffer shared with the target Worker. This operating mode has been chosen because the middlebox (and then the Master itself) is supposed to be traversed by a huge amount of traffic; hence, a blocking model would be too penalizing because it would require an interrupt-like mechanism to start the Master whenever new data are available. This could significantly drop the performance with high packet rates [28]. In fact, interrupt handling is expensive in modern superscalar processors because they have long pipelines and support out of order and speculative execution [29], which increase the penalty paid by an interrupt.

Vice versa, since the traffic entering into a specific Worker is potentially a small portion compared to the one handled by the Master, a blocking model looks more appropriate for this module. This ensures the possibility to share CPU resources more effectively, which is important in multi-tenant systems where potentially a large number of Workers is active. Hence, when a Worker has no more data to be processed, it suspends itself until the Master wakes it up by means of a shared semaphore.

3.3. Basic algorithm: handling pass-through data

The algorithm used to move data from the Master to the Workers (and back) requires the sharing of some variables (underlined in the pseudocode shown in the following), a semaphore, and the primary buffer between the Master and each Worker. In particular, in this section we assume the presence of the Master and a *single* Worker, while its extension to several Workers is trivial.

The primary buffer is operated through four indexes. `M.prodIndex` and `W.prodIndex` are shared between the Master and the Worker. The former index points to the next empty slot in the buffer, ready to be filled by the Master, while the latter points to the next slot in the buffer that the Worker will make available to the Master again after its processing. `M.prodIndex` is incremented by the Master when it enqueues new tokens, while `W.prodIndex` is incremented by the Worker when it makes processed tokens available to the Master again. `M.consIndex` is a private index of the Master and points to the next token that the Master itself will remove from the buffer. Finally, `W.consIndex` is a private index of the Worker and points to the next token to be processed by the Worker itself. In addition to these indexes, the algorithm exploits the shared variable `workerStatus`, which indicates whether the Worker is suspended or it is running.

Algorithm 1 provides the overall behavior of the Master and shows how it cyclically repeats the following three main operations. First, in lines 14-21 it produces new data (line 19), which corresponds to the reception of packets from the network interface card (NIC) in the NFV use case, and immediately provides them to the Worker through the primary buffer (line 20). Second, it reads the tokens already processed by the Worker

from the primary buffer (line 22). Third, it wakes up the Worker if there are data waiting for service for a long time in order to avoid starvation (line 23). From lines 14-21, it is evident that the Master produces several tokens consecutively, in order to better exploit cache locality. Furthermore, if the buffer is full (line 15), it stops data production and starts removing the tokens already processed by the Worker from the buffer.

Algorithm 1 Executing the Master

```
 1: Procedure master.do()
 2:
 3: {Initialize shared variables}
 4: M.prodIndex ← 0
 5: W.prodIndex ← 0
 6: workerStatus ← WAIT_FOR_SIGNAL
 7:
 8: {Initialize private variables of the Master}
 9: M.consIndex ← 0
10: timeStamp ← 0
11:
12: {Execute the algorithm}
13: while true do
14:    for i = 0 to (i < N or timeout()) do
15:       if M.prodIndex == (M.consIndex−1) then
16:          {The buffer is full}
17:          break
18:       end if
19:       data ← master.produceData()
20:       master.writeDataIntoBuffer(data)
21:    end for
22:    master.readDataFromBuffer()
23:    master.checkForOldData()
24: end while
```

Algorithm 2 details the mechanism implemented in the Master to send data to the Worker. As shown by line 8, a token is inserted into the slot pointed by the shared index `M.prodIndex` as soon as it is produced. However, the Worker is awakened only if at least a given number of tokens (i.e., `MASTER_PKT_THRESHOLD`) are waiting for service in the primary buffer (lines 10-13). Thanks to this threshold, we avoid to wake up the Worker for each single token that needs to be processed, which results in performance improvement because (*i*) it reduces the number of context switches and (*ii*) it increases cache locality, for both data and code. Since a token is inserted into the buffer as soon as it is produced (regardless of the fact that the Worker is running or not), and since the Worker will suspend itself only when the buffer is empty (as detailed in Algorithm 5), the Worker is able to process a huge amount of data consecutively, thus improving system performance.

Our algorithm avoids the starvation of tokens sent to a Worker (which may happen especially when

Algorithm 2 The Master writing data into the primary buffer

```
1: Procedure master.writeDataIntoBuffer(Data d)
2:
3: if M.prodIndex == M.consIndex then
4:    {The buffer is empty}
5:    timeStamp ← now()
6: end if
7:
8: buffer.write(M.prodIndex,d)
9: M.prodIndex++
10: if buffer.size() > MASTER_PKT_THRESHOLD and
    (workerStatus ≠ SIGNALED) then
11:    workerStatus ← SIGNALED
12:    wakeUpWorker()
13: end if
```

the system is in underload conditions) thanks to a timeout event that wakes up the Worker even if the above-mentioned threshold is not reached yet. In particular, the Master acquires and stores the current time whenever it inserts a new token and the buffer is empty (lines 3-6 of Algorithm 2). This way, the Master knows the age of the oldest token and it is able to possibly wake up the Worker also depending on the value of a given time threshold, as shown in Algorithm 3.

Algorithm 3 The Master waking up the Worker due to a timeout

```
1: Procedure master.checkForOldData()
2:
3: if buffer.size() > 0 and (workerStatus ≠ SIGNALED) and
    ((now() − timeStamp) > TS_THRESHOLD) then
4:    workerStatus ← SIGNALED
5:    wakeUpWorker()
6: end if
```

The functions described in Algorithm 2 and Algorithm 3 need to know whether the Worker is already running or not in order to avoid useless Worker awakenings. This information is carried by the shared variable `workerStatus`, which is set to `SIGNALED` by the Master just before waking up the Worker (line 11 of Algorithm 2 and line 4 of Algorithm 3), and changed to `WAIT_FOR_SIGNAL` by the Worker just before suspending itself (line 22 of Algorithm 5). This way, the Master can test this shared variable to have an indication about the Worker status, and then wake it up only when necessary.

Algorithm 4 shows how the Master removes from the primary buffer the data that have already been processed by the Worker. In particular, it consumes all the tokens until the index `M.consIndex` does not reach the index `W.prodIndex`, incremented by the Worker each time it has handled a batch of tokens, as detailed in Algorithm 5. In this way, also the Master reads several consecutive data from the primary buffer in order to better exploit cache locality.

Algorithm 4 The Master reading data from the primary buffer

```
 1: Procedure master.readDataFromBuffer()
 2:
 3: if buffer.size() then
 4:    if M.consIndex ≠ W.prodIndex then
 5:       timeStamp ← now()
 6:       while M.consIndex ≠ W.prodIndex do
 7:          if not buffer.dropped(M.consIndex) then
 8:             master.consumeData(buffer.read(M.consIndex))
 9:          end if
10:          M.consIndex++
11:       end while
12:    end if
13: end if
```

Notice that Algorithm 4 also considers those tokens provided by the Master to the Worker, and dropped by the Worker itself. In case of dropped data, the Master receives back an empty slot, identified through the flag `dropped`. The content of a slot is only consumed if this flag is zero, otherwise the Master just increments the `M.consIndex` and moves on to the next slot of the buffer, as shown in lines 7-10. This prevents the Master from reading a slot with a meaningless content.

Algorithm 5 details the operations of the Worker. As evident from lines 12-23, whenever a Worker wakes up, it processes all the tokens available in the primary buffer (i.e., all the slots of the buffer with indexes less than `M.prodIndex`). Only at this point (line 24), as well as after it has processed a given amount of data (lines 13-16), the Worker updates the shared index `W.prodIndex`, so that the Master can consume all the tokens already processed by the Worker itself. This way, the Master will be notified for data availability only when a given amount of tokens are ready to be consumed, with a positive impact on performance. It is worth noting that this batching mechanism is different from the one implemented when the Master sends data to the Worker. In fact, in that case, the Worker is woken up when the amount of data into the buffer is higher than a threshold, although the `M.prodIndex`, used by the Worker to understand when it has to suspend itself, is incremented each time a new data is inserted. Here, instead, the `W.prodIndex` (i.e., the index used by the Master to know when the consuming of tokens must be stopped) is not updated each time the Worker processes a data. As a consequence, it is possible that some tokens have already been processed by the Worker, but the `W.prodIndex` has still to be updated, and then the Master cannot consume them in the current execution of Algorithm 4. This results in a slightly higher latency for these tokens, but in better performance for the system thanks to this batching processing enabled into the Master. As a final remark, lines 18-20 show that the Worker can drop the token under processing by setting the `dropped` flag in the current slot of the primary buffer.

Algorithm 5 Executing the Worker

```
1: Procedure worker.do()
2:
3: {Initialize private variables of the Worker}
4: W.consIndex ← 0
5: pkts_processed ← 0
6:
7: {Execute the algorithm}
8: while true do
9:    waitForWakeUp()
10:   W.consIndex ← W.prodIndex
11:   pkts_processed ← 0
12:   while W.consIndex ≠ M.prodIndex do
13:      if pkts_processed == WORKER_PKT_THRESHOLD then
14:         pkts_processed ← 0
15:         W.prodIndex ← W.consIndex
16:      end if
17:      toBeDropped ← buffer.process(W.consIndex)
18:      if toBeDropped then
19:         buffer.setDropped(W.consIndex)
20:      end if
21:      W.consIndex++
22:      pkts_processed++
23:   end while
24:   W.prodIndex ← W.consIndex
25:   workerStatus ← WAIT_FOR_SIGNAL
26: end while
```

Figure 3 depicts the status of the primary buffer[1] and the indexes used by the algorithm in four different time instants. In Figure 3(a) the buffer is empty, and then all the indexes point to the same position. Instead, in Figure 3(b) the Master has already inserted some data into the buffer, but the Worker is still waiting since the `MASTER_PKT_THRESHOLD` has not been reached yet. Figure 3(c) depicts the situation in which the Master has woken up the Worker, which has already processed two items. Notice that, since the `WORKER_PKT_THRESHOLD` has not been reached yet, the `W.prodIndex` still points to the oldest token in the buffer. Instead, in Figure 3(d) this threshold is passed and the Master has already consumed some data.

3.4. Extended algorithm: handling Worker-generated data

Our algorithm also supports Workers that may need to generate *new* data as a consequence of the token just received from the Master. However, this cannot be done with the primary buffer alone, as Workers cannot *inject* new data into the primary buffer. The Worker can in fact just modify (potentially completely) pass-through tokens in the primary buffer or, at most, it can drop these tokens.

Then, in case new data have to be provided to the Master, the Worker can use the auxiliary buffer. This buffer, in which the Worker acts as the producer while the Master plays the role of the consumer, is

[1]For the sake of clarity, the figure represents the shared buffer as an array instead of a circular FIFO queue.

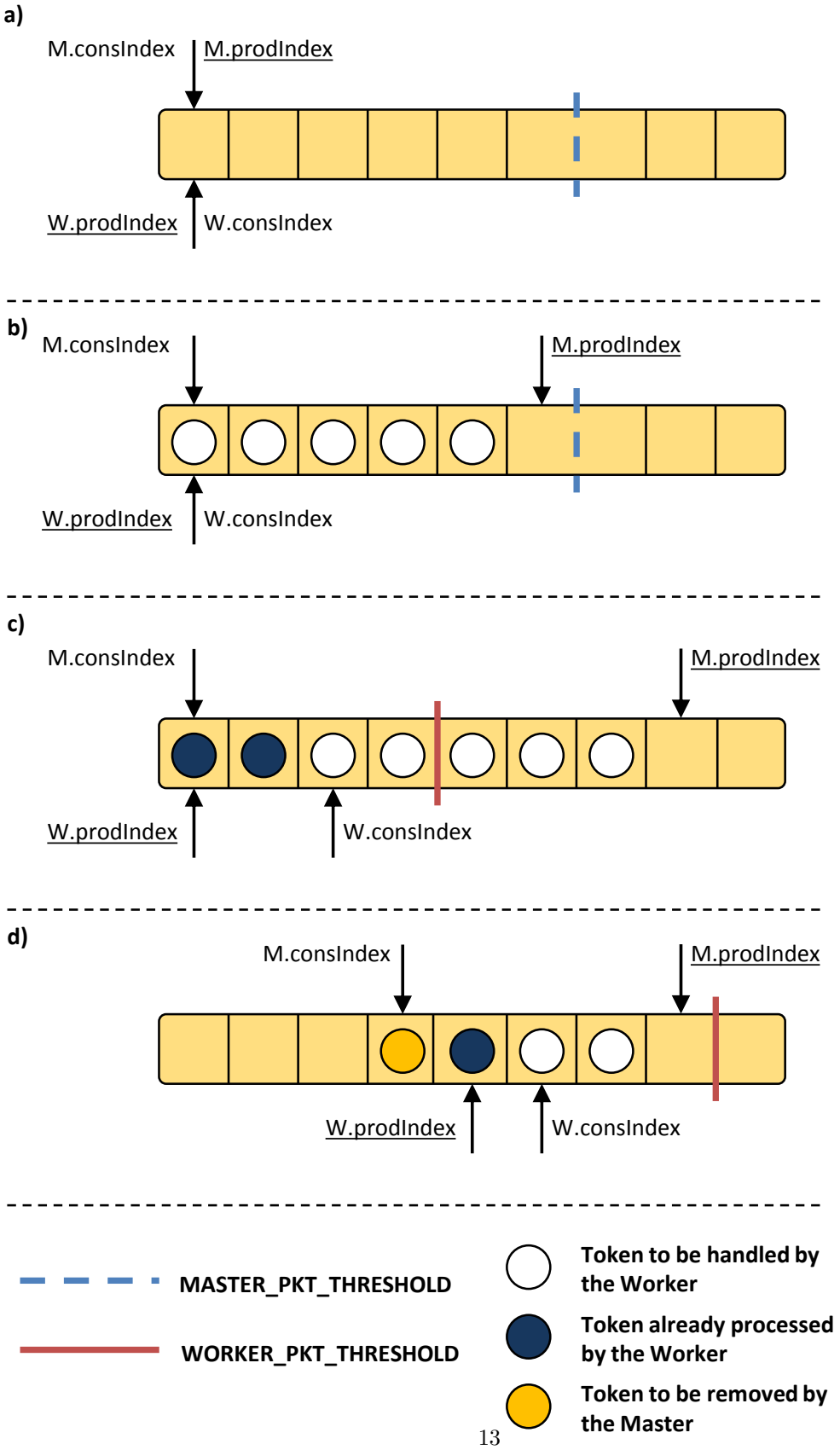


Figure 3: Run-time behavior and indexes of the algorithm.

managed through two indexes; moreover, it requires a further flag in each slot of the primary buffer, which indicates whether the next token should be read from the primary or the auxiliary buffer.

Algorithm 6 details how the Worker sends new data to the Master, as a consequence of the processing of the token at position `W.consIndex` in the primary buffer. As shown in lines 3-11, several data can be generated for a single token received from the Master, and all these data are then linked to the same slot of the primary buffer. A first flag, called `aux`, is set in the slot of the primary buffer to signal to the master that the next slot to read is the one on top of the auxiliary buffer (line 13). Instead, the `next` flag set in a slot of the auxiliary buffer tells that the next packet has still to be read from the auxiliary buffer, instead of returning to the next slot of the primary buffer.

Algorithm 6 The Worker writing *new* data into the auxiliary buffer

```
 1: Procedure worker.writeDataIntoAuxBuffer(Data[] newData, Index W.consIndex)
 2:
 3: while data ← newData.next() do
 4:    if auxProdIndex == (auxConsIndex-1) then
 5:       {The auxiliary buffer is full}
 6:       break
 7:    end if
 8:    auxBuffer.write(auxProdIndex,data)
 9:    auxBuffer.setNext(auxProdIndex)
10:    auxProdIndex++
11: end while
12: auxBuffer.resetNext(auxProdIndex-1)
13: buffer.setAux(W.consIndex)
```

The reading procedure is described in Algorithm 7. When the Master encounters a slot with the `aux` flag set in the primary buffer, it processes a number of tokens in the auxiliary buffer, starting from the slot pointed by `auxConsIndex` until the `next` flag is set. Moreover, according to lines 4-7 of Algorithm 6, if the auxBuffer is full, new tokens that the Worker may want to send to the Master are dropped.

Algorithm 7 The Master reading data from the auxiliary buffer

```
 1: Procedure master.readDataFromAuxBuffer()
 2:
 3: while true do
 4:    master.consumeData(auxBuffer.read(auxConsIndex))
 5:    if not auxBuffer.next(auxConsIndex) then
 6:       auxConsIndex++
 7:       break
 8:    end if
 9:    auxConsIndex++
10: end while
```

Figure 4 depicts the primary buffer with some slots linked to the auxiliary buffer. In particular, the slot pointed by `M.consIndex` is associated with two data of the auxiliary buffer, i.e., the one pointed by

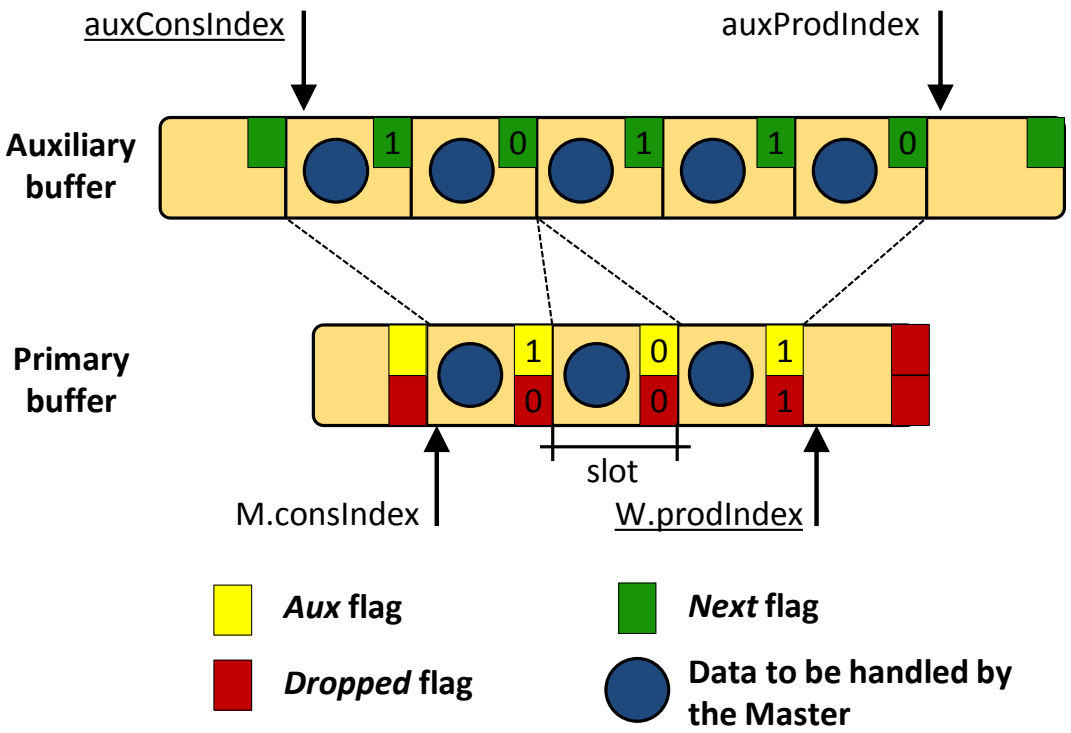


Figure 4: Binding primary buffer - auxiliary buffer.

`auxConsIndex` and the following one. This second slot has then the `next` flag reset, in order to indicate that the next slot is not linked with the current slot in the primary buffer. Instead, the next token in the primary buffer is not associated with the secondary buffer (the `aux` flag is reset), while the third slot contains data dropped by the Worker, althouth it is linked to three data in the auxiliary buffer. In other words, the configuration in which `aux == 1` and `dropped == 1` is valid and it enables to completely replace a packet with a new one.

4. Formal verification

Assessing the correctness of an algorithm is often not straightforward, hence we built an abstract model of the Master with a single Worker in order to formally check some fundamental properties. We do not consider a plurality of Workers because the interaction between the Master and a Worker is independent of the interaction with any other Worker, hence this approach is sufficient to demonstrate the correctness of the whole system. In particular, we only focus on the primary buffer, as its operation is one of the main contributions of our work and hence it needs a proof of correctness. Instead, the auxiliary buffer is not explicitly verified as it is managed as a standard producer/consumer system, which has been already studied and validated in the existing literature [30].

The model of our algorithm has been developed in Promela [31], a well known modeling language that, in conjunction with the SPIN [32] model checker generator, can be used to formally verify distributed and concurrent software against certain desired properties. The main purpose of the model checking technique is to explore all the possible states of a system and verify whether the specified properties hold in *each* execution path. Whenever the model checker finds an execution path that leads to a property violation, it provides the full counter-example with all the steps needed to reach the undesired behavior.

When creating an accurate model of the system, it is very important to keep the nature of the problem *tractable*, as model checking verification tools tend to exploit a massive amount of memory (state-space explosion problem). Therefore, the actual model of the data exchange mechanism has been built by omitting some implementation details that are not relevant for the analyzed properties, in order to reduce the overall number of states. This is possible because many system states (or runs) are mapped to the same abstract state (or run). A more detailed description of our model will be provided in Section 4.2.

4.1. Properties specification

Given the structure of our algorithm, we can identify six properties that *must* be always satisfied. The first two properties refer to conditions on some key variables that must be verified to guarantee that no slots will be erroneously overwritten, formally defining regions of the buffer that are "owned" by one of the two modules (the Master and the Worker) at a given time.

Property 1. *`W.prodIndex` must never exceed `M.prodIndex`.*

`M.prodIndex` indicates the first empty position in the primary buffer that must be filled by the Master. Hence, it represents a boundary that the Worker must never pass.

Property 2. *`M.consIndex` must never exceed `W.prodIndex`.*

`M.consIndex` represents the position of the token being processed by the Master, while `W.prodIndex` identifies the position of the last token that can be processed by the Master.

We also consider two additional safety properties that must be satisfied by the system. Specifically we require that:

Property 3. *The number of pending tokens delivered by the Master to the Worker and not yet processed by the Worker itself is, at any time, a non negative integer not exceeding the maximum number of elements that the buffer can store, namely (N - 1), where N is the total buffer size:*

$$0 <= tokens\_master\_to\_worker <= (N-1)$$

Property 4. *The number of pending tokens delivered by the Worker to the Master and not yet processed by the Master itself is, at any time, a non negative integer not exceeding the maximum number of elements*

that the buffer can store, namely (N - 1), where N is the total buffer size:

$$0 <= tokens\_worker\_to\_master <= (N-1)$$

Our circular buffer implementation always leaves at least one empty position, so as to distinguish the cases where the buffer is completely full or completely empty. This is why the actual buffer capacity is N-1.

Finally, we consider two more properties related to the overall system behavior.

Property 5. *Deadlock absence.*

This property means that neither the Master nor the Worker ever enter an infinite waiting situation.

Property 6. *Livelock absence.*

This last property ensures that some useful work is eventually done by the Master. Here the notion of "useful work" is intended as the Master capability to produce, sooner or later, new tokens for the Worker, e.g., by inserting new data into the buffer. This is an important property verified under the assumption that some fairness constraints are satisfied. Fairness constraints are necessary when the model includes nondeterministic choices, in order to ensure that these choices are made in a fair way; this means that, in each run, it must not be possible that the nondeterministic choice always gives the same result. For example, the scheduling of processes is modeled by Spin using nondeterminism. As usual, we assume the process scheduler always gives the possibility to both the Master and the Worker to execute, sooner or later, some instructions. This is a reasonable hypothesis since most modern execution environments implement scheduling algorithms to avoid process starvation.

Details about how these properties have been specified in Promela are provided in section 4.2.4.

4.2. Model details

4.2.1. The primary buffer

Our abstract model does not require the modeling of realistic data into the buffer, but only the modeling of the buffer status. Consequently, only the indexes, the maximum buffer size (i.e. N), and the actual buffer size (i.e. the number of tokens currently stored in the buffer, denoted $buffer\_size$ in the Promela model) are modeled.

4.2.2. The semaphore and the functions

The model of the semaphore consists in an asynchronous channel shared between the Master and the Worker. Basically, the *blocking wait* operation is modeled by reading a constant from the channel, while the *signaling* operation is modeled by writing the same constant into the channel. This is a very common

pattern, useful to model various kinds of communication/synchronization primitives between two or more entities.

The functions presented in the pseudocode in Section 3 are modeled as Promela processes, since the language does not provide an explicit way to represent functions and their returned value. We exploit the following well known pattern: the caller sends a token through a synchronous channel shared with the callee in order to pass the control to the invoked process. Then, it performs a *receive* operation on the same channel in order to be awakened from the other end-point when the processing has terminated. Notice that the channel can also be used to pass arguments to and from the called process/function.

4.2.3. The Master and the Worker

The two main entities, the Master and the Worker, are modeled as two independent, concurrently running processes. They share the `M.prodIndex` and `W.prodIndex` variables and the channel/semaphore (as stated in our pseudo-code in Section 3.3), in addition to the buffer status variables. In order to decrease the amount of states to be verified by the model checker, and hence reduce the overall verification time to a reasonable value, we use the following abstractions. First, in the if-statement of Algorithm 3, the check on the timestamp value is replaced in the model by a nondeterministic choice, as the whole model does not contain any explicit information about the elapsing of time. Second, the `timeout()` function that is present in the loop guard (Algorithm 1) is replaced by a non-deterministic choice. This means that, rather than modeling a realistic mechanism to implement a timeout event, we instructed the model checker to extract a random value to decide whether a timeout has occurred or not. Both these abstractions provide a significant performance enhancement without any loss in terms of exhaustiveness of the verification.

4.2.4. The Properties

In properties 1 and 2, the condition that one index must not exceed another index has to be expressed taking into account that, in a circular buffer, indexes are reset to zero when they reach the end of the buffer. For this reason, it would be wrong to simply state property 1 as `W.prodIndex` $\leq$ `M.prodIndex`. Actually, after having been reset to zero, `M.prodIndex` has to be smaller than or equal to `W.prodIndex` until `W.prodIndex` is reset to zero too. According to this consideration, this property is expressed by means of a conditional assertion, the condition being a boolean state variable (named `work_index_M_prod_index_inequality`) that is flipped whenever one of the two counters is reset to zero. The conditional assertion is written in Promela as follows:

```
if
  :: (work_index_M_prod_index_inequality == 0); assert (W_prod_index <= M_prod_index);
  :: (work_index_M_prod_index_inequality == 1); assert (W_prod_index >= M_prod_index);
```

```
fi;
```

For checking properties 3 and 4 we use a counter that represents the number of outstanding tokens (from Master to Worker for property 3 and from Worker to Master for property 4). This counter is incremented when a new token is produced and decremented when a token is consumed. At each increment or decrement operation, an assertion is introduced in order to check that the counter value remains within its intended range.

Property 5 (deadlock absence) is a built-in property automatically checked by Spin, so no specification is necessary for it.

Properties 1-5 are all safety properties. They can be checked all together by Spin, via a single reachability analysis. Property 6 is instead a liveness property that can be expressed by means of the following Linear Temporal Logic (LTL) formula:

$$(fairness\_constraint) \Rightarrow []((master\_progress == 0) \Rightarrow (<> (master\_progress == 1))) \quad (1)$$

This formula takes the form of an implication, where the left hand side ($fairness\_constraint$) is another LTL formula specifying the additional fairness constraints[2] that are assumed for the analysis of the right hand side of the implication. The latter is expressed in terms of the *master_progress* boolean variable which tracks the execution of useful work done by the Master. At the beginning of each loop of the master scheduling algorithm, *master_progress* is set to 0, whereas it is set to 1 when the Master produces some new data for the Worker. The property specifies that it is always true ([]) that, if *master_progress* is 0, eventually (<>) it will become 1, thus expressing the fact that the Master continually executes useful work (i.e. produces new data).

The additional fairness constraints $fairness\_constraint$ are related to the two nondeterministic choices used in the model to decide whether a timeout has occurred, and whether the Worker has to be signaled because the tokens produced by the Master have become too old. They are expressed by the following LTL formula, composed of two different sub-constraints:

$$[]((scheduling == 0) \Rightarrow <> (scheduling == 1)) \;\&\&\; []((old\_flag == 0) \Rightarrow <> (old\_flag == 1)) \quad (2)$$

The first sub-constraint involves the boolean variable *scheduling* that records, at each master loop, whether the timeout has been triggered ($scheduling = 0$) or not ($scheduling = 1$). This first constraint specifies

[2]Thes constraints are in fact in addition to the standard constraint about the scheduling of processes, which can be automatically considered by Spin.

	Parameters			
	BUFFER SIZE	MASTER THRESHOLD	WORKER THRESHOLD	**VERIFICATION RESULT**
Property 1	[1,8]	[1,8]	[1,8]	SUCCESS
Property 2	[1,8]	[1,8]	[1,8]	SUCCESS
Property 3	[1,8]	[1,8]	[1,8]	SUCCESS
Property 4	[1,8]	[1,8]	[1,8]	SUCCESS
Property 5	[1,8]	[1,8]	[1,8]	SUCCESS
Property 6	[2,8]	[1,8]	[1,8]	SUCCESS

Table 1: Algorithm verification.

that, starting from any time instant, eventually there will be an iteration of the loop in which the timeout event does not occur. The second constraint is similar, but it involves the boolean variable old_flag, which records the nondeterministic result of the check on the timestamp value in the if-statement of Algorithm 3.

In order to avoid unnecessary complexity, property 6 has been checked separately, in a run where safety properties are not checked. The two full Promela models (one for checking safety properties and the other one for checking property 6) are publicly available in [33].

4.3. Verification results

The model explained above can be exhaustively verified for different values of the main model parameters, as shown in Table 1. For each property, the table specifies the considered range of values for the buffer size, the `MASTER_PKT_THRESHOLD` and the `WORKER_PKT_THRESHOLD`. For the sake of scalability of the verification process and without losing in generality, we used rather small values compared to a realistic buffer, which could contain millions of tokens. In fact, possible structural bugs would be detected also in a small size system deployment.

Some inconsistent parameters settings in the considered ranges, such as a threshold greater than the buffer size, are skipped in our verification work. Notice also that, with a buffer size equal to one token, Property 6 is not considered, as the buffer cannot contain any token and therefore the master is not able to perform any useful work. Properties 1-5 are verified even without forcing any fairness constraint, since their satisfaction depend neither on how processes are scheduled, nor on the nondeterministic choices that model the time-related aspects.

In conclusion, the results of our verification process completely demonstrate the correctness of the algorithm from different points of view (absence of indexes misbehavior or accidental packets overwriting, and absence of deadlocks and livelocks). These results can be reproduced using the models available in [33].

5. Implementation

The achievable performance depends not only on design, but also on implementation issues. Then this section presents some implementation choices that can improve performance and scalability of our algorithm, and that have been adopted in our prototype implementation.

Private copies of shared variables. As in many algorithms derived from the producer-consumer problem, also in our case we need to keep two processes in sync by means of a pair of shared variables, one written only by the first process, the other one written only by the second process. Although in this case concurrency issues are limited (no contention can occur because the two processes never try to write the same variable at the same time), the actual implementation on real hardware can introduce additional issues, as shown in MCRingBuffer [17]. In fact, when a first CPU core modifies the content of a variable that is shared with a different CPU core, the entire cache line (64 bytes long on the modern Intel architectures) of the second core containing that variable is invalidated. If the second core needs to read that variable, the hardware has to retrieve this value either from the shared cache (e.g., the L3 in many recent Intel architectures) or from the main memory, with a consequent performance penalty. In our algorithm, this problem occurs for `M.prodIndex`, incremented by the Master whenever a new token is inserted into the primary buffer and read by the Worker, and for `W.prodIndex`, incremented by the Worker and read by the Master. However, our algorithm is robust enough to operate correctly even if those variables are not perfectly aligned. As a consequence, the Worker creates a private copy of `M.prodIndex` just after waking up, while the Master copies in a private variable the content of `W.prodIndex` before reading data from the shared buffer. The Master and the Worker can perform their operations according to the value of their local copies, which are re-aligned with the actual values only periodically; this does not preclude the correct system operation while ensuring a significant reduction of cache misses.

Shared variables on different cache lines. Because of the same problem mentioned in the previous paragraph (a CPU core can invalidate an entire line of cache in the other cores), our code implements a cache line separation mechanism (similar to MCRingBuffer [17]) that consists in storing each shared variable (possibly extended with padding bytes) on a different cache line. This way, when the Master changes, for instance, the value of `M.prodIndex`, the cache line containing `W.consIndex` is not invalidated in the private cache of the core where the Worker is executed.

Alignment with cache lines. In case of a cache miss, the hardware introduces a noticeable latency because of the necessity to transfer the data from the memory to the cache, which happens in blocks of fixed size (the *cache line*). From that moment, all the memory accesses within that block of addresses are

very fast, as data are served from the L1 cache. In order to minimize the number of cache misses (and the associated performance penalty), our prototype was engineered to align the most frequently accessed data so that they span across the minimum set of cache lines. In particular, the starting memory address of the packet buffers and their slot sizes are multiple of the cache line size; the same technique is used for minimizing the time for accessing the most important data used in the prototype.

Use of huge memory pages. Huge pages are convenient when a large amount of memory is needed, since they decrease the pressure on the Translation Lookaside Buffer (TLB) for two reasons. First, the load of virtual-to-real address translation is split across two TLBs (one for huge pages and the other for normal memory), preventing normal applications (based on normal pages) from interfering with the packet exchange mechanism (which uses huge pages). Second, they reduce the number of entries in the TLB when a large amount of memory is needed. We use the huge pages for the shared (primary and auxiliary) buffers; the drawback is the potential increase of the total memory required by the algorithm because the minimum size of each buffer increases from 4KB to 2MB.

Preallocated memory. Dynamic memory allocation should be avoided during the actual packet processing, as this would heavily decrease the performance of the whole system. In our case, all the buffers used by the packet exchange mechanisms are allocated at the startup of each Worker, allowing the system to add/remove workers at run-time while at the same time avoiding dynamic memory allocation.

Emulated timestamp. Getting the current time is usually rather expensive on standard workstations, as it requires the intervention of the operating system and, often, an I/O operation involving the hardware clock. In our case, we emulate the timestamp needed to wake up a Worker when packets are waiting for service for too long time, by introducing the concept of *current round*, that is the number of loops executed by the Master in Algorithm 1. As a consequence, our implementation schedules a Worker for service when there are packets waiting for more than N rounds, where N is a number that can be tuned at run-time based on the expected load on the Master.

Batch processing. Batch processing is convenient because it keeps a high degree of code and data locality, with a positive impact on cache misses. Our prototype implements batch processing whenever possible, e.g., the Master reads all waiting packets from a Worker before serving the next, and Workers process all the packets in their queue before suspending themselves. The drawback is the potential increase of the latency in the data transfer.

Semaphores. A simple POSIX semaphore is used to wake up a Worker in case at least `MASTER_PACKET_THRESHOLD` packets are queued in the primary buffer, or in case some packets are waiting for long time and then the

timeout expired. Although POSIX semaphores are implemented in kernel space, their impact on performance is acceptable as they are rarely accessed by algorithm design. Instead, no explicit signal is used in the other direction: the shared variables `M.consIndex` and `W.prodIndex` are in fact used by the Master to detect the presence of packets that need to be read from the buffer.

Threading model. Context switching should be avoided whenever possible because of its cost, particularly when this event happens frequently (such as in packet processing applications, which are usually rather simple and often handle a few packets in a row). For this reason, the Master is a single thread process, cycling on a busy-waiting loop and consuming an entire CPU core, while Workers (which are single-thread processes as well) work in interrupt mode and share the remaining CPU cores. While the Master can be simply parallelized over multiple cores as long as the function chains are not interleaved[3], by design our implementation keeps it locked to a single core. In fact, we would like to allocate the most part of the processing power to the (huge number of) Workers, as they host the network functions that are in charge of the actual (useful, from the perspective of the end users) processing.

6. Experimental results

In order to evaluate performance and scalability of the data exchange algorithm described in Section 3, we carried out several tests on our prototype implementation (that follows the implementation choices described in Section 5) running on a workstation equipped with an Intel i7-3770 @ 3.40 GHz (four CPU cores plus hyperthreading), 16 GB RAM, 16x PCIe bus, a couple of Silicom dual port 10 Gigabit Ethernet NICs based on the Intel x540 chipset (8x PCIe), and Ubuntu 12.10 OS, kernel 3.5.0-17-generic, 64 bits. In all tests, an entire CPU core is dedicated to the Master, while Workers have been allocated on the remaining CPU cores in a way that maximizes the throughput of the system. All the following graphs are obtained by averaging results of 100s tests repeated 10 times.

The data exchanged among the Master and the Workers consists of synthetic network packets of three sizes: 64 bytes to stress the forwarding capabilities of the chain, 700 bytes that matches the average packet size in current networks, and 1514 bytes to stress the data transfer capabilities of the system. We first present a set of experiments where packets exchanged between the Master and the Workers are directly read/written from/to the memory, without involving the network. These tests aim at validating the performance of the algorithm in isolation, without any disturbance such as the cost introduced by the driver used to access to

[3]Interleaved chains may introduce additional complexity because multiple Masters may collide when feeding a single Worker. This would then require an extension of our algorithm (no longer lock-free) that is left to a future work.

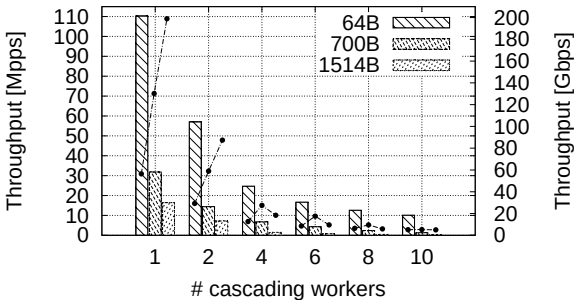
the NIC or the overhead of the PCIe bus. In these testing conditions, Section 6.3 compares our algorithm against two existing approaches based on the traditional producer/consumer paradigm, which are typically used to exchange packets between the vSwitch and the network functions consolidated on the same server. Particularly, the comparison shows the advantages deriving from both the absence of data copy in the Worker and the blocking operating mode of the Worker itself. Finally, Section 6.6 presents some results involving a real network, where the workstation under test is connected with a second workstation acting as both traffic generator and receiver through two 10Gbps dedicated NICs. This setup allows to derive the precise latency experienced by packets in our middlebox. In this case we use the PF_RING/DNA drivers [26] to read/write packets from/to the NIC, since they allow the Master to send/receive packets without requiring the intervention of the operating system. In addition, data coming from the network is read in polling mode in order to limit additional overheads due to NIC interrupts, and in batches of several packets in order to maximize code locality. Similar techniques are used also when sending data to the network after all the processing took place.

6.1. Single chain - Throughput

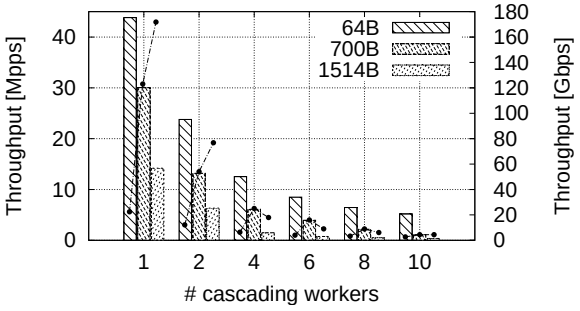
This section reports the performance of our algorithm in a scenario where all packets traverse the same chain, which is statically defined. Tests are repeated with chains of different lengths and the measured throughput is provided in graphs that include: *(i)* a bars view corresponding to the left Y axis that reports the throughput in millions of packets per second; *(ii)* a point-based representation referring to the right Y axis, which reports the throughput in Gigabits per second.

Figure 5 shows the throughput offered by the function chain in different conditions. These numbers depend both on the design aspects of our algorithm (e.g., no data copy in the Worker, polling model in the Master, blocking model in the Worker, etc.), as well as on the implementation choices we did when implementing the prototype (e.g., data aligned with cache lines, private copies of shared variables, etc., as detailed in Section 5). For instance, the overall throughput of the chain (i.e., the packets/bits that exit from the chain) decreases with the number of Workers because of our choice of reserving the most part of the CPU power to the Workers, hence limiting the Master to a single CPU core (Section 5 - threading model).

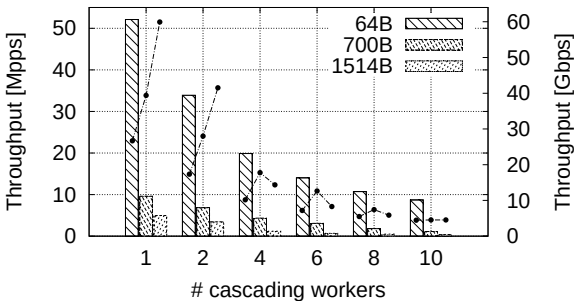
Figure 5(a) shows the throughput that could be achieved in ideal conditions, that is: *(i)* with dummy Workers, i.e., Workers that do not touch the packet data, and *(ii)* with the Master always reading the same input packet from memory and copying it into the buffer of the first Worker of the chain. This in fact reduces the overall number of CPU cache misses experienced at the beginning of the chain and provides an ideal view of the system, where the penalties due to memory accesses are kept to a minimum. Results reported



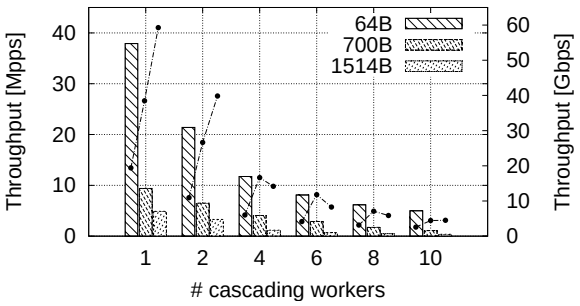
(a) Dummy Workers and a single packet in memory.



(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.

Figure 5: Throughput of a single function chain with the algorithm presented in this paper.

in Figure 5(b) are instead gathered in a more realistic scenario, i.e., with Workers that access to the packet content and calculate a simple signature across the first 64 bytes of packets. This may represent a realistic workload, as it emulates the fact that most network applications operate on the first bytes (i.e., the headers) of the packet. This test shows that performance is reduced compared to Figure 5(a) for two reasons: (*i*) the higher number of cache misses generated by the Workers when accessing to the packet content, and (*ii*) the additional processing time spent by the Workers for completing their job.

Next tests consider a scenario where the input data for the chain is stored in a buffer containing 1M packets, thus emulating a real middlebox that receives traffic from the network. In particular, Figure 5(c) refers to a scenario with dummy Workers such as in Figure 5(a) and shows how an apparently insignificant different memory access pattern can dramatically change the throughput. In fact, the Master experiences frequent cache misses when reading packets at the beginning of the chain. This modification alone halves the throughput compared to Figure 5(a), particularly when packets have to traverse chains of limited length. Instead, in case of longer chains, this additional overhead at the beginning is amortized by the cost of the rest of the chain.

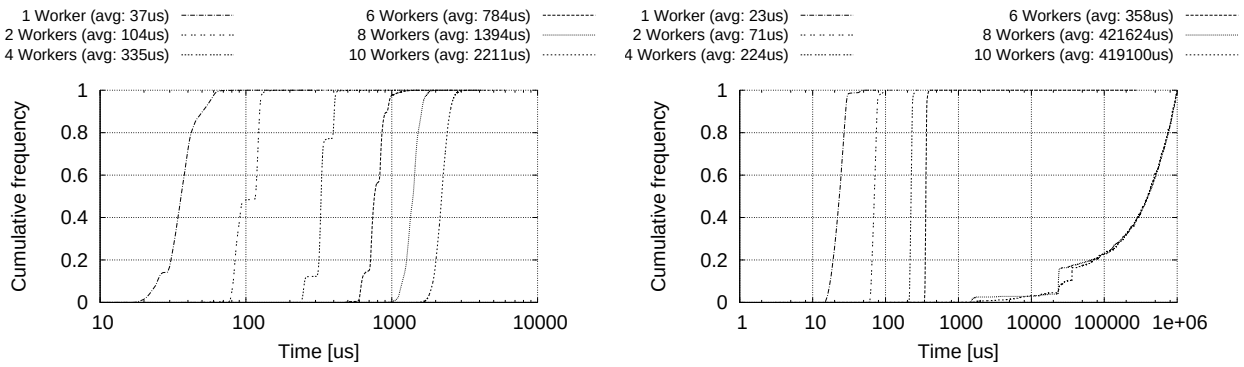
Finally, Figure 5(d) depicts a realistic scenario where Workers access the packet content (such as in Figure 5(b)), and the Master feeds the chain by reading data from a large initial buffer (1M packets). Even in this case our algorithm is able to guarantee an impressive throughput, such as about 38 Mpps with 64B packets.

In order to confirm that, with the current workload, the Master represents the bottleneck of the system, Figure 7 shows the internal throughput of the chain, namely the total number of packets moved by the Master in the same test conditions of Figure 5(d). This figure gives an insight of the processing capabilities of the Master, which slightly increase with a growing number of Workers, and proves the effectiveness of our algorithm as the number of packets it processes essentially does not depend on the number of Workers.

6.2. Single chain - Latency

Some architectural and implementation choices, such as working with batches of packets, aim at improving the throughput but may badly affect the latency. For this reason, this section gives an insight about the latency experienced by packets traversing our chains. Measurements are based on the `gettimeofday` Unix system call and, in order to reduce its impact on the system, only sampled packets (one packet out of thousand) have been measured.

Figure 6(a) shows the latency of 64B packets when traversing a function chain consisting of a growing number of Workers, in case of real Workers and 1M packets in memory. As expected, the latency increases



(a) Our algorithm.

(b) Zero-copy buffers among the Master and the (polling) Workers.

Figure 6: Latency introduced by the function chain with a growing number of cascading Workers.

with the length of the chain; however its value is definitely reasonable for most of networking applications, reaching an average value of about 2.2ms in case of 10 cascading Workers, being far less with shorter (and more realistic) chains.

6.3. Single chain - Comparison with other algorithms

This section aims at comparing our data exchange algorithm with two other approaches that could be used to exchange packets between the Master and the Workers, and which represent the baseline algorithms used to evaluate the improvements (in terms of performance) brought by our research. Particularly, the comparison aims at validating the advantages of two important aspects of our algorithm: the absence of a data copy in the Worker, and the blocking operating mode of the Worker itself.

In this respect, we cannot directly compare our algorithm with existing prototypes used in NFV such as VALE [15], OVS [21] and xDPd [22], because they include the overhead of packet classification (e.g., L2 forwarding, Openflow matching), which would affect the performance of the data exchange algorithm. As a consequence, we distilled the fundamental design choices of the most important alternative approaches and we carefully implemented them by using, whenever applicable, the guidelines listed in Section 5 (e.g., shared variables on different cache lines, private copies of shared variables, and more).

The first baseline algorithm is based on the traditional producer/consumer paradigm, in which the Master shares two buffers with each Worker: the first is used by the Master to provide packets to the Worker, while the second operates in the opposite direction. In this case, similarly to our algorithm, only the Master operates in polling mode, while the Worker wakes up when there are packets to be processed. The second baseline algorithm closely follows the processing model suggested by Intel in the DPDK library [11]. Also in

this case two buffers (again based on the traditional producer/consumer paradigm) are shared between the Master and each Worker; however, these buffers contain pointers, which means that the actual data is stored in a shared memory region and never moved between the components of the functions chain (zero-copy). Moreover, both the Master and Workers operate in polling mode. Although this solution neither provides isolation among the Workers, nor limits the CPU consumption, it has been selected as a baseline algorithm to be compared against our proposal because nowadays it represents the "standard" way to move packets in network function chains.

The baseline algorithms are executed in realistic conditions, namely with Workers accessing packets and 1M packets in memory; therefore, obtained results should be compared with numbers reported in Figure 5(d).

As expected, the throughput of the chain drops of about 30% when unidirectional buffers are used, as shown by comparing Figure 5(d) and Figure 8(a). This is mainly due to the operating principles of our primary buffer, which allows the Worker to send back a packet to the Master without moving the packet itself, while the baseline algorithm requires one additional data copy in the Worker.

Instead, the second baseline algorithm slightly outperforms our algorithm until the number of jobs (one Master plus N Workers) is lower than the number of available CPU cores, as evident by comparing Figure 8(b) with Figure 5(d). This is due to the absence of data copies and to the polling-based operating mode implemented in the Workers. However, a stronger performance degradation with respect to our solution (it offers less than 1 Mpps throughput) is noticeable when 8 (or more) Workers are active, because at least two of them have to share the same CPU core.

The second baseline algorithm has also been evaluated in terms of latency introduced on the flowing packets. Similarly to what happens for the throughput, it outperforms our proposal when the number of running jobs is less than the number of CPU cores, as evident by comparing Figure 6(a) and Figure 6(b).

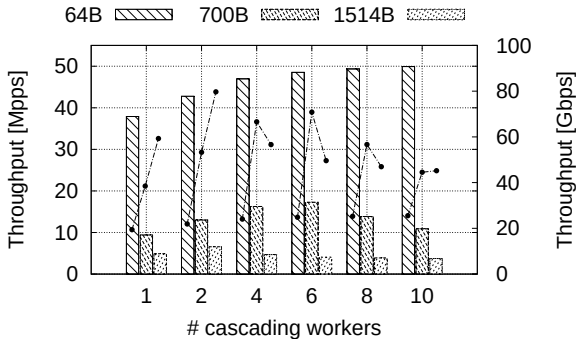
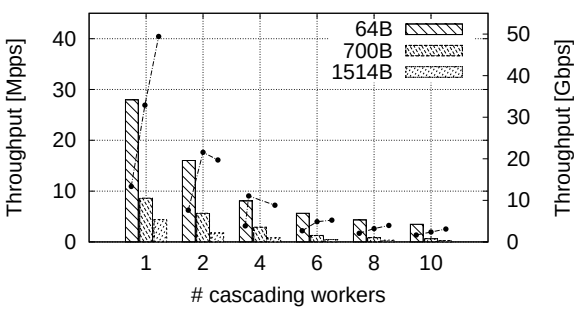
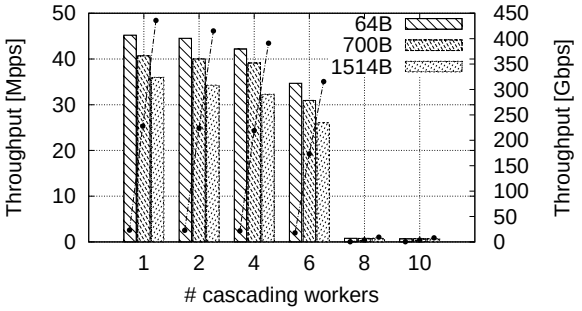


Figure 7: Internal throughput of the function chain, with real Workers and a 1M packets in memory.



(a) Unidirectional buffers shared between the Master and the Workers.



(b) Zero-copy buffers among the Master and the (polling) Workers.

Figure 8: Throughput of a single function chain when other data exchange algorithms are used.

For instance, six chained Workers introduce an average latency of 358μs, against the 784μs obtained with our algorithm. Instead, in case of more Workers, the average latency of the baseline algorithm reaches 420ms, which is definitely not acceptable. This poor performance is due to the fact that many polling processes share the same CPU core. Hence, this solution neither provides isolation among Workers (due to the zero-copy), nor acceptable performance when the number of Workers exceeds the number of available cores, being inappropriate for our objectives.

6.4. Single chain - Other tests

Additional tests have been performed in order to evaluate some other aspects of the system.

6.4.1. Threads vs. Processes

Threads appear more convenient than processes because they share the same virtual memory space, while processes have distinct virtual memory spaces. In our system, where the data exchange mechanism requires a shared memory between the Master and a Worker, this could have an impact on both the cache efficiency and the TLB behavior and, consequently, on the overall performance of the system. With respect to the former, two processes sharing the same physical memory address have two virtual addresses, which requires two entries in the L1/L2 caches[4]. On the contrary, threads have the same virtual address, hence potentially allow the same cache line to be used by different threads. With respect to the TLB, as the same (virtual) address space is present in many threads, the number of entries in the TLB is reduced as well. Instead, processes are expected to generate an higher number of TLB misses.

[4]The L3 cache operates with physical addresses.

In order to guarantee memory isolation among Workers, which is a key point in a multi-tenant NFV node, the Master and all the Workers should be implemented as different processes. Although this suggests a possible performance penalty compared to the thread-based implementation, our experiments dismantle this belief, as the overall performance is definitely similar in both cases. The reason is that the L1/L2 caches are private per each physical core, but the Master and the Workers are usually executed in different cores. Hence, an address already cached by the core executing the Master cannot be already found in the cache of the core executing the Worker, forcing the latter to retrieve that data from the (physically addressed) L3 cache, no matter whether it is a thread or a process. As a consequence, as far as performance is concerned, our system shows no differences between a thread-based and a process-based implementation.

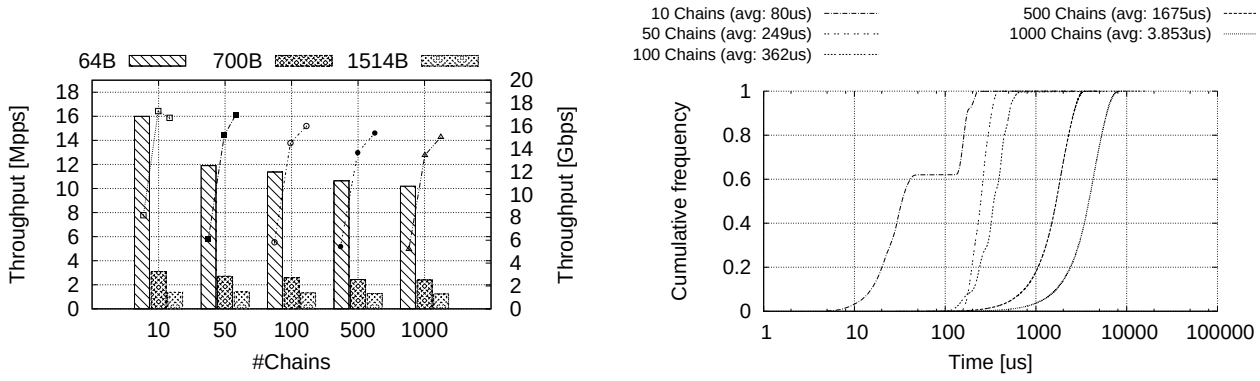
6.4.2. Normal memory vs. huge pages

We also evaluated the impact of our choice of using huge pages (each one consisting of 2MB of memory in our testbed) instead of normal pages (4KB) for the shared buffers. Although it may sound strange, results of the two approaches do not differ significantly in the test scenarios considered so far. This is a consequence of our specific test conditions, where the Master and the Workers use a very little amount of memory in addition to the shared buffers. Hence, we repeated the test with Workers executing a deep packet inspection algorithm based on a Deterministic Finite Automata (DFA), which requires a huge amount of memory to store the DFA used to recognize the given patterns into the packets. In this case, the adoption of the huge pages for the shared buffer results in roughly a 10% improvement in terms of throughput.

6.5. Multiple chains

While previous tests focused on packets traversing a growing number of functions all belonging to the same chain, this section evaluates the case when multiple function chains are executed in parallel and each packet traverses only one of them. This significantly stresses the CPU cache, as *(i)* the Master has to receive 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.

Data read from the initial memory buffer (containing 1M packets) is provided, in a round robin fashion, to a growing number of function chains, each one composed of two Workers. During the tests, each Worker is involved in two chains meaning that, when 1000 Workers are deployed, packets are spread across 1000 different function chains. Workers are allocated among six CPU cores in a way that minimizes the number of times a packet has to be moved from one core to another, in order to limit CPU cache synchronization operations among cores (Section 5).



(a) Throughput.

(b) Latency.

Figure 9: Results with a growing number of function chains running in parallel, each one spotting two Workers in cascade.

Obtained results are shown in Figure 9. As in the previous tests, these numbers are due to the combined effect of the choices we did when designing our algorithm (Section 3) and of the implementation guidelines followed to efficiently implement the prototype (Section 5).

Figure 9(a) provides the overall throughput measured at the end of all the chains, which smoothly decreases with the increment of the number of chains. Notably, it is equal to several Gbps also with 1000 chains in the system, thus confirming the effectiveness of our algorithm. Figure 9(b) shows instead the cumulative distribution of the latency experienced by 64B packets traversing the chains, which ranges from an average value of 80μs in case of 10 function chains, to an average value of 3.8ms when 1000 chains are active.

6.6. Network tests

This section evaluates our algorithm in a real deployment scenario, i.e., when executed on a workstation that receives/sends traffic from the network. In this case the overall performance of the system depends on the algorithm, on the implementation choices done when developing the prototype, as well as on additional aspects such as the driver used for accessing the NIC. Anyway, these results provide an insight of the behavior of the algorithm when used in the context it was designed for.

The throughput obtained in this scenario, whose testing conditions are the same as those of Figure 5(d), is depicted in Figure 10(a). Results are limited by the speed of the input NIC in several cases, particularly with large packets and (relatively) short chains. With longer chains (i.e., 10 cascading workers) the throughput is even slightly better than what was obtained in Figure 5(d) without the network. This can be due to the fact that real NICs create an input buffer that is much smaller than the 1M packets buffer used in the previous

(a) Throughput.

(b) Latency.

Figure 10: Results with a function chain of growing length, with the Master accessing to the network.

test, hence potentially improving the data locality.

Figure 10(b) shows the cumulative distribution function of the latency introduced by network function chains of different length when traversed by 64B packets. Those numbers, obtained by sending packets at the same rate shown in Figure 10(a), measure the time between the instant in which the packet is scheduled for transmission in the traffic generator, and the time it is received by our testing software in the traffic receiver. In this case we then consider all the time spent by the packet in our middlebox, plus the network latency and the time spent in the traffic generator/receiver after/before hitting our timestamping code. Particularly, reported numbers also include the time that the packet spends in the input buffer before being picked up and sent through the chain by the Master, because of its batch-based reading mode. Our measurements demonstrate that the latency, albeit still acceptable, is about 4-5 times higher than in Figure 6(a).

7. Conclusion

This paper has proposed an efficient way to move data between network functions (the Workers) and a virtual switch module (the Master), in order to implement virtual network function chains. The architecture is based on a different pair of circular buffers shared between the Master and each Worker, and aims at achieving a scalable and high performance system while guaranteeing traffic isolation among the different (huge number of) Workers.

One of the peculiarities of this approach is that, through the primary buffer, data are sent to a Worker and then returned back to the Master for further processing with zero-copy. A form of batching has also been introduced in order to amortize the cost of context switches, while introducing a safeguard mechanism

to avoid packet starvation in case of Workers traversed by a limited amount of traffic. The auxiliary buffer is instead used by the Worker to send new data to the Master.

Formal verification techniques have been applied in order to rigorously prove the absence of deadlocks and livelocks, and also to guarantee that no packet can be accidentally overwritten due to concurrency issues such as race conditions or incorrect use of shared indexes.

Finally, performance and scalability of the proposed solution have been evaluated by means of a wide range of experiments made on a real implementation.

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] European Telecommunications Standards Institute, Network Functions Virtualisation, White paper, SDN and OpenFlow World Congress, Darmstadt, Germany (Oct. 2012).

[2] F. Risso, I. Cerrato, Customizing data-plane processing in edge routers, in: Proceedings of the First European Workshop on Software Defined Networking (EWSDN), 2012, pp. 114–120. doi:10.1109/EWSDN.2012.14.

[3] I. Cerrato, G. Marchetto, F. Risso, R. Sisto, M. Virgilio, An efficient data exchange algorithm for chained network functions, in: High Performance Switching and Routing (HPSR), 2014 IEEE 15th International Conference on, 2014, pp. 98–105.

[4] L. Zhao, L. Bhuyan, R. Iyer, S. Makineni, D. Newell, Hardware support for accelerating data movement in server platform, Computers, IEEE Transactions on 56 (6) (2007) 740–753. doi:10.1109/TC.2007.1036.

[5] M. M. Michael, 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, PODC '96, ACM, New York, NY, USA, 1996, pp. 267–275. doi:10.1145/248052.248106.

[6] A. Gidenstam, H. Sundell, 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, OPODIS'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 302–317.

[7] S. Prakash, Y. H. Lee, T. Johnson, A nonblocking algorithm for shared queues using compare-and-swap, IEEE Trans. Comput. 43 (5) (1994) 548–559. doi:10.1109/12.280802.

[8] M. Hoffman, O. Shalev, N. Shavit, The baskets queue., in: E. Tovar, P. Tsigas, H. Fouchal (Eds.), OPODIS, Vol. 4878 of Lecture Notes in Computer Science, Springer, 2007, pp. 401–414.

[9] H. Massalin, C. Pu, Threads and input/output in the synthesis kernal, in: Proceedings of the twelfth ACM symposium on Operating systems principles, SOSP '89, ACM, New York, NY, USA, 1989, pp. 191–201. doi:10.1145/74850.74869.

[10] H. Massalin, C. Pu, A lock-free multiprocessor os kernel, SIGOPS Oper. Syst. Rev. 26 (2) (1992) 108–.

[11] Intel, Data plane developer kit - programmers guide (2012).

[12] P. Tsigas, 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, SPAA '01, ACM, New York, NY, USA, 2001, pp. 134–143. doi:10.1145/378580.378611.

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

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

[15] L. Rizzo, G. Lettieri, Vale, a switched ethernet for virtual machines, in: Proceedings of the 8th international conference on Emerging networking experiments and technologies, CoNEXT '12, ACM, New York, NY, USA, 2012, pp. 61–72. doi:10.1145/2413176.2413185.

[16] J. Hwang, K. K. Ramakrishnan, 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), USENIX Association, Seattle, WA, 2014, pp. 445–458.
[17] P. P. C. Lee, T. Bu, G. P. Chandranmenon, A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring, in: IPDPS, 2010, pp. 1–12.
[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, SIGOPS Oper. Syst. Rev. 37 (5) (2003) 164–177. doi:10.1145/1165389.945462.
[19] K. K. Ram, A. L. Cox, M. Chadha, S. Rixner, Hyper-switch: A scalable software virtual switching architecture, in: Proceedings of the 2013 USENIX Conference on Annual Technical Conference, USENIX ATC'13, USENIX Association, Berkeley, CA, USA, 2013, pp. 13–24.
[20] S. Govindan, J. Choi, A. Nath, A. Das, B. Urgaonkar, A. Sivasubramaniam, Xen and co.: Communication-aware cpu management in consolidated xen-based hosting platforms, Computers, IEEE Transactions on 58 (8) (2009) 1111–1125. doi:10.1109/TC.2009.53.
[21] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, S. Shenker, Extending networking into the virtualization layer, in: Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII), 2009.
[22] xdpd.
URL `http://www.xdpd.org`
[23] J. Blendin, J. Rückert, N. Leymann, G. Schyguda, D. Hausheer, Position paper: Software-defined network service chaining, in: Proceedings of the Third European Workshop on Software Defined Networking (EWSDN 2014), 2014.
[24] I. Cerrato, T. Jungel, A. Palesandro, F. Risso, M. Suñé, H. Woesner, User-specific network service functions in an sdn-enabled network node, in: Proceedings of the Third European Workshop on Software Defined Networking (EWSDN 2014), 2014, pp. 135–136.
[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74.
[26] F. Fusco, L. Deri, High speed network traffic analysis with commodity multi-core systems, in: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC '10, ACM, New York, NY, USA, 2010, pp. 218–224. doi:10.1145/1879141.1879169.
[27] L. Rizzo, Netmap: a novel framework for fast packet i/o, in: Proceedings of the 2012 USENIX conference on Annual Technical Conference, USENIX ATC'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 9–9.
[28] J. C. Mogul, K. K. Ramakrishnan, Eliminating receive livelock in an interrupt-driven kernel., in: USENIX Annual Technical Conference, 1996, pp. 99–112.
[29] C. Dovrolis, B. Thayer, P. Ramanathan, Hip: Hybrid interrupt-polling for the network interface, ACM Operating Systems Reviews 35 (2001) 50–60.
[30] S. Doherty, L. Groves, V. Luchangco, M. Moir, Formal verification of a practical lock-free queue algorithm, in: D. de Frutos-Escrig, M. Nez (Eds.), Formal Techniques for Networked and Distributed Systems FORTE 2004, Vol. 3235 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 97–114.
[31] G. Holzmann, Spin Model Checker, the: Primer and Reference Manual, 1st Edition, Addison-Wesley Professional, 2003.
[32] G. J. Holzmann, The model checker spin, IEEE Transactions on Software Engineering.
[33] GitHub, Shared buffer model.
URL `https://github.com/netgroup-polito/shared-buffer`