UCLouvain

epl

École polytechnique de Louvain

SmartNIC-Based Scheduling

oRSS : An offloaded packet scheduler

Author: **Clément Delzotti**
Supervisor: **Tom Barbette**
Readers: **Etienne Riviere, Nikita Tyunyayev**
Academic year 2022–2023
Master [120] in Computer Science

Abstract

Following Moore's law slowing down this last decade, single CPU cores aren't able to match always increasing network capacities anymore. In this situation, sharding imposes itself as a very scalable solution to fill this performance gap but requires flows to be processed by the same cores each time. The literature focused on finding a way to provide such characteristics while providing a balanced distribution between cores to maximize the available performances and reduced resource consumption. But with the emergence of SmartNICs, new opportunities open as hardware offloading becomes accessible. This document proposes oRSS, a packet scheduler that leverages SmartNIC offloading to provide a balanced flow distribution among cores based on processor utilization while maximizing the CPU cycles available for application processing. We prove that such a scheduler is possible and can provide better performances than well-known techniques such as RSS with a throughput gain that can reach 20% on average.

Contents

Chapter 1

Acknowledgements

This work wouldn't have been possible without the help of many people. I'd like to thank :

Tom Barbette, my supervisor, for his precious advice during this past year and his weekly follow-up.

Nikita Tyunyayev, Tom's assistant, for his technical help and his reviews.

Etienne Riviere, UCLouvain professor, for accepting to read this document and be a member of the jury.

Guillaume Nizet, Martin Danhier and Vincent Higginson, my friends, who made up for the absolute nonexistence of my visualization skills.

Chapter 2

Introduction

Over recent years, datacenters bandwidth's impressive increase coupled with rather stagnant evolutions of other components such as processors created a gap between network capabilities and hosts processing power, at the disadvantage of the latter [3]. A single CPU core isn't capable of handling the entire network load anymore, opening the way to aggressive multicore techniques to cope with network capacity.

From all these techniques, one appears to be prominent : *Sharding*[1]. It consists of running each core in a completely isolated manner, without any intercore communication or synchronization. It only requires splitting the workload across cores. In the case of computer networks, for stateful processing reasons, it must ensure that a given flow is always handled by the same core, through what is called *flow-to-core affinity*[1][10].

Literature already contains multiple propositions corresponding to this description, such as *RSS* [10] and *RSS++* [1]. The first dispatches flows randomly across cores, possibly overloading some cores while under loading others. The latter presents itself as an improvement that applies a dynamic balancing depending on each flow's CPU consumption to avoid load imbalances. While *RSS++* successfully balances flows across shards to reduce CPU consumption, it requires the host to collect counters maintained by each core and compute an image of current core utilization. This computation nibbles processor cycles that therefore cannot be dedicated to application processing. Additionally, it works with buckets of flows that must not be too large.

At the same time, Network Interface Cards have evolved to be more programmable, allowing to offload some packet processing from the host to the NIC itself. Such NICs are called *SmartNICs*. Through the combination of multiple hardware accelerators [15], those NICs are capable to alleviate hosts from some of

their control logic, gaining precious CPU cycles to application processing.

Different initiatives have tried to compensate for RSS static scheduling by leveraging the programmability of SmartNICs. Elastic RSS [22] uses a programmable NIC to schedule packets across multiple *scaling groups*. But this comes at the cost of a dedicated CPU core on the host to allocate cores to scaling groups, focusing on providing adequate cores to the whole application instead of reducing imbalances between cores loads. Once again, this consumes resources that cannot be dedicated to application processing anymore. To the best of our knowledge, there isn't any initiative attempting to schedule flows based on their CPU utilization in order to maximize data processing resources while taking advantage of SmartNICs offloading yet.

This document presents *oRSS*, a packet scheduler that takes advantage of both sharding and SmartNIC offloading to provide such scheduling without requiring any dedicated core on the host. It replicates RSS behaviour and implements a balancing mechanism, allowing flow migrations between cores to redistribute them in a fairer manner. This document describes its architecture and compares its performances with *RSS*.

To put the results of our tests in a nutshell, oRSS has the advantage over RSS that it doesn't have to get lucky, leading to a throughput increase of up to 60% and a per-core imbalance reduced to up to a fourth. In our test configuration, oRSS quickly matches up its slight overhead to get the advantage of a more balanced flow distribution, ensuring a better throughput than RSS with a reduced number of migrations.

In Chapter 2, we'll investigate the state of literature for different concepts oRSS relies upon: *Handling kernel overhead*, *SmartNICs*, *flow management*, *Virtual switching* and of course *Packet scheduling*. The following chapter explains how oRSS mobilizes this knowledge into an offloaded CPU-load-based scheduler. Then, in Chapter 4, we'll dig into oRSS performances through different use cases. Finally, the last chapter concludes with discussions about this work's contribution to the literature.

Chapter 3

Background

3.1 Overview

This section intends to describe the state of the different disciplines oRSS relies upon. We separate them into 5 categories: *Kernel overhead*, *SmartNICs*, *Flow management*, *Virtual switching* and finally *Packet scheduling*.

The kernel overhead section (3.2) describes the limitations of the current Linux network stack and explores different solutions [3]. These solutions can be inside the kernel itself, such as AF_XDP [12], or imply a complete bypass through netmap or DPDK [4][14]. Alternatively, some solutions even propose an alternative kernel[2][21][19]. These solutions are essential as they allow overcoming unsustainable overheads that made the Linux network stack a bottleneck [3] that keeps it from matching modern network performances.

The following section digs into the SmartNIC topic. It first describes the importance of SmartNICs as they provide kernel interposition [23]. We'll then see how a load-balancer [5] can be implemented through these programmable NICs. To conclude this section, the architecture of two important SmartNICs is inspected in detail: Netronome NFP [13] and Nvidia Bluefield [15].

A short section 3.4 goes over important considerations concerning flow sizes, underlining the importance of discerning elephants and mice flows [8] in order to build a viable scheduler.

The penultimate section 3.5 dives in the important subject of virtual switching, through OpenVSwitch [20] and OpenFlow [16].

Finally, we'll examine different packet schedulers proposed by literature, such as RSS [10] and RSS++ [1]. Some solutions already provide offloaded packet

schedulers like eRSS [22]. It is certainly very close to the work described in this document, but unlike oRSS it focuses on scaling available CPU cores instead of providing a balanced dispatching of flows among cores. Moreover, it requires a dedicated core on the host, confiscating important computing resources for control logic that therefore cannot be dedicated to data processing. Other papers [24] [28] describe more generic approaches to packet scheduling. Finally, we'll discuss informed request scheduling on the NIC [9].

These thematics are of paramount importance to understand how oRSS works. It leverages kernel bypass to circumvent Linux network stack bottleneck, allowing a linerate of 100Gbps. As its name suggests, oRSS leverages offloading on a SmartNIC, freeing important CPU computation power. We decided to apply this offload on an Nvidia Bluefield-2, which is an off-path NIC whose embedded switch can be controlled through an OpenVSwitch instance [18] itself controllable through OpenFlow. Finally, oRSS combines all these technologies to provide an efficient packet scheduler.

3.2 Kernel Overhead

3.2.1 Host Network Stack Overheads

Over the past few years, data centers access link bandwidth has kept increasing while essentially all other host resources have been stagnant. To answer that challenge, many efforts have been made, such as RDMA and hardware offloads. In such a context, understanding the inefficiencies of the traditional Linux networking stack is of paramount importance. These inefficiencies [3] are :

- **Data copy becomes the main overhead**: In a modern Linux network stack, the main bottleneck shifts from protocol processing to data copy. Copying data from kernel buffers to application buffers can take more than 50% of CPU cycles for a single long flow. This problem is even more important on the receiving side, where cross-NUMA node data copies can lead to higher CPU consumption.
- **Direct Cache Access can lead to suboptimal throughput**: Direct Cache access allows NICs to write packets directly into L3 caches. As host processing becomes the bottleneck, it increases host latencies. This leads to packets writing in L3 outpacing application data copy of cached data, causing high cache miss rates and up to 24% drop in throughput-per-core.
- **Host resource sharing can be harmful**: Any shared resources between flows is a source of performance degradation, as it necessarily implies lower

cache hit rates and hinders the capacity for a host to gain for certain optimizations (Coalescing packets, etc.).

- **CPU aware scheduling is important**: Both long and short flows generate different overheads. Long flows generate a high data copy overhead, while short flows are causing a high packet processing overhead. Consequently, having different types of flows on the same CPU core can reduce performance. A CPU scheduler that is aware of application types can efficiently schedule long and short flows to maximize performance.

3.2.2 In-kernel solutions

To face Linux network stack overhead, the first possibility is to leverage in-kernel solutions such as AF_XDP.

AF_XDP

AF_XDP [12] takes advantage of the *eXpress Data Path* eBPF Hook to allow users to specify kernel behaviour when a packet is received. It has the advantage of providing the same socket abstraction as the classical network stack to user space applications. Users must provide an XDP program that will be in charge of redirecting packets to the right socket. To avoid constantly copying packets to user space, AF_XDP allocates incoming packets to a memory area allocated by the user application. Instead of transmitting a complete packet, it only copies a descriptor indicating packet location and length to the user space.

Performances

While bringing significant improvements to the Linux network stack, AF_XDP sockets performances are below those offered by complete kernel bypass solutions such as DPDK. But it has both the advantage of working through a kernel abstraction (i.e., sockets) and is available to a wider range of hardware thanks to different XDP modes.

3.2.3 Kernel-Bypass Techniques

We saw in section 3.2.1 that kernel forms a bottleneck, limiting the capacity of hosts to take advantage of all the bandwidth at their disposal. To face this problem, a solution can be to circumvent this kernel stack. This section investigates these bypass possibilities [4].

User space I/O

The first solution is to retrieve packets directly from the NIC to the user space. This can be done with *netmap* by allowing user space applications to directly retrieve data from kernel drivers (therefore avoiding system calls and extra copying) or with *DPDK* by running the NIC drivers directly in user space that can be polled frequently by the application (therefore bringing the same advantages as netmap without driver interruptions). While this can bring a significant increase in the bitrate directly available to applications, it leaves to the user space the charge to reimplement any kernel feature it may need (ARP, IP verifications, TCP reliability, etc.). Other solutions such as RDMA [11] allow network hosts to write directly into the user space memory of other hosts. This has the benefit of not only bypassing the kernel but also performing I/O operations without involving any action from the receiver's CPU.

User space TCP/IP stack

A more compromised solution is a user space TCP/IP stack to bypass the kernel while still providing TCP/IP features to applications. Examples are **mTCP** and **IX**, further described in section 3.2.4.

Hybrid network stack

A classical user space stack can lack features and doesn't benefit from the Linux stack's heavy development. A hybrid solution is possible, which consists in porting the full kernel TCP/IP stack to fast user space I/O processing. This can be done with **StackMap** which leverages netmap.

3.2.4 Changing the kernel

As Kernel-Bypass is widely adopted to circumvent kernel overhead, the abstraction between user applications and the network gets thin. In a situation where applications take direct advantage of hardware features and accelerators, it hinders the capacity of hardware to evolve without impacting numerous applications. This problem adds up to application complexity as they must handle their own network logic.

Demikernel

Counterintuitively, after getting around the kernel abstractions, kernel-bypass might actually lack some abstractions. A solution to this problem can be a *Demikernel* [29]. As figure 3.1 shows, it is divided into two distinct paths: control and data. The first

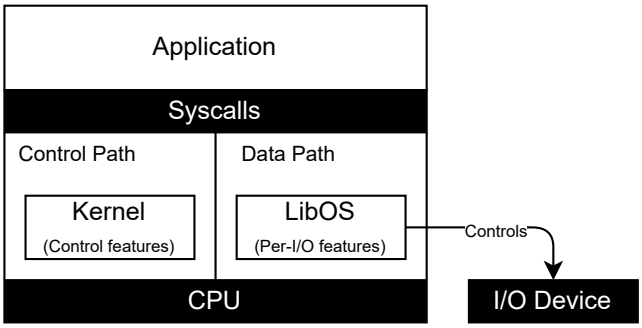


Figure 3.1: Diagram representing the Demikernel architecture, making explicit the difference between the data path and control path.

gathers all OS features that are not used on a per-I/O basis, including kernel-bypass accelerator allocations to applications and connection opening, while the latter provides OS functionalities that directly handle I/O, including reading and writing to hardware devices. Features provided by this Demikernel architecture are such:

- **I/O queues** : An abstraction of kernel-bypass I/O devices. Applications push and pop data to and from I/O devices through high-level queues. This requires an atomic data unit to know how much data should be sent to the hardware or retrieved from it.
- **System call interface** : Defines interactions between applications and the Demikernel. Classical system calls that would usually return a file descriptor now return a *queue descriptor*. The most important data path operations are *push* and *pop* to send and receive data. Syscalls are also used to handle control path features such as queue creation or connection establishment.
- **Other features** such as memory management and event/thread scheduling can also be provided by the Demikernel architecture.

IX

Data centers applications usually suffer from a 4-way trade-off between :

- **Low tail latency**: As any user request can involve hundreds of servers, the tail latency becomes important as it will be experienced by many requests. Thus, a low tail latency is important to allow interactions between many services without significantly increasing the latency experienced by users.
- **High packet rates**: Requests and sometimes replies between services of a data center can be small, generating a need for a high packet rate.

- **Protection**: In a situation where multiple services share servers, the need for application isolation arises. This can be enforced through kernel or hypervisor-based networking stack.
- **Resource efficiency**: As the load on data centers applications varies a lot during diurnal patterns, generating spikes in user traffic, it is important to have applications that scale in terms of consumed resources.

Modern server hardware allows such low latency and packet rates, but operating systems have been designed under restrictive assumptions (multiple applications sharing a single processing core, inter-arrival times being higher than system calls latency, etc.). This leads to operating systems trading latency and throughput in favour of better resource scheduling.

IX [2] aims at providing an OS architecture that fits the efficiency criteria described above. To do so, it follows some key principles :

- **Separating and protecting control and data plane** such that (in opposition to what is usually achieved through kernel bypass, where different planes mix up).
- **Run to completion with adaptive batching** to avoid intermediate buffering between stages and reduce latency.
- **Native zero-copy** to avoid the classical bottleneck due to data copy.
- **Flow consistent processing** via RSS to ensure consistent mapping of incoming traffic to distinct hardware queues.

Evaluation
IX demonstrates that it is possible to implement a protected OS kernel while still delivering linerate performances. It noticeably outperforms Linux and mTCP in terms of latency and throughput.

Head-Of-Line blocking
But IX doesn't answer the problem of Head-Of-Line blocking. In a situation where loads are subject to high variance, head-of-line blocking can arise. A CPU-heavy flow might keep a core busy for a while and thus keeps CPU-light flows waiting in the queue. ZygOS [21] achieve a μ-scale tail-latency by implementing preemption.

Shenango

Although kernel bypass can achieve micro-second scale latency, it is quite far to be CPU-efficient [19][14]. Notably because of its poll-based packet detection and its lack of core scaling, leading to constantly busy CPU cores and a core allocation that must be able to handle peak load at all times.
A proposed solution is *Shenango* [19], aiming at reallocating cores across applications at very fine time scales by leveraging existing Linux facilities. It detects load changes every five μs and performs core allocations at rates such as 60000 times per second.

Shenango takes advantage of an *IOKernel* that serves as an intermediary between applications and the NIC. It runs on a single busy-spinning core, bypassing the kernel. Thanks to this approach, Shenango is capable of quickly steering packets between cores and access queue states. It approximates core usage based on these queue states, in an attempt to detect congestion. As discussed in 3.4, this keeps Shenango from identifying precisely which flow is responsible for the congestion, forcing a coarse scaling per group of flows.

3.3 SmartNICs

Data centers are embracing SmartNICs [5]. They provide cost-effective computing for end-host network functionalities. These SmartNICs embed cheap but weak multicore processors, assisted with packet manipulation and cryptographic accelerators, allowing some advanced packet processing before it reaches the host [5][15][13]. Effectively offloading a process onto a SmartNIC requires addressing some challenges [5] :

- A SmartNIC is equipped with weak cores and limited memory, therefore it isn't suitable for general-purpose computing.
- As it is weakly geared, it requires some lightweight synchronization to take full advantage of the multicore structure of the embedded processor.
- The weak cores on their own will simply create a bottleneck. Efficient use of a SmartNIC requires leveraging different accelerators (Packet transformation, Cryptography, etc) to propose decent performances.

SmartNICs are usually structured as either *On-Path* or *Off-Path*, as in figure 3.2 [5]. In the first case, every packet goes through the NIC cores. As these cores' performances are limited, this implementation requires hardware support such as prefetching or hardware-assisted packet buffer management to keep up. In the second case, only a few packets are going through the NIC cores, while most are

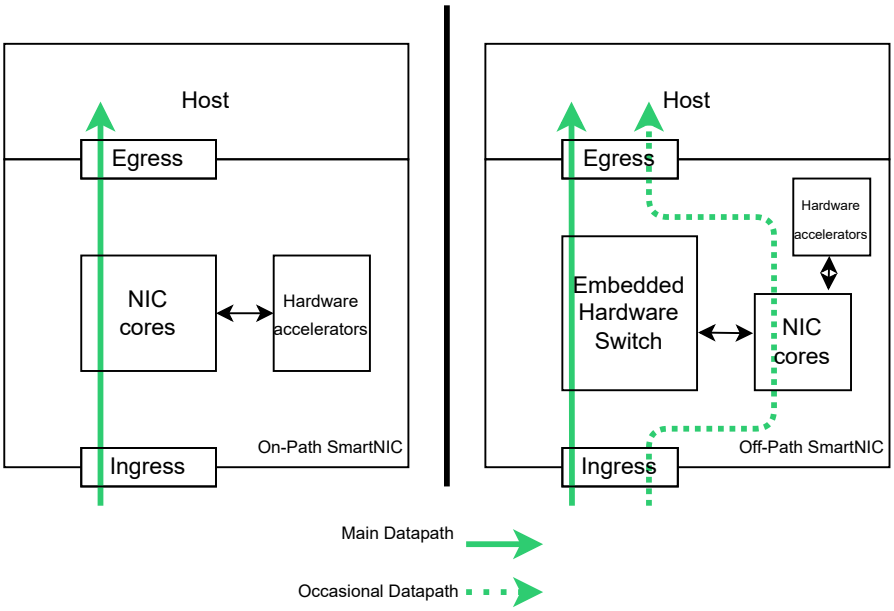


Figure 3.2: Illustration of the difference between On-Path and Off-Path SmartNICs. On-path implies that every packet goes through NIC's cores before being sent to the host. On the other hand, Off-Path SmartNIC sends most packets through a hardware switch that is controlled by NIC's cores.

going through a NIC switch to directly reach the host. It thus leaves to the NIC processor the management of this switch.

By bringing programmability closer to the network, SmartNICs allow various optimizations [15] :

- **Reduced host resource utilization**: A SmartNIC allows moving workloads from the host to the NIC, therefore relieving the host of such tasks.
- **Accelerated specialized operations**: Some SmartNICs embed specialized hardware to accelerate particular operations (e.g., encryption).
- **Improved power efficiency**: By typically working with ARM CPUs and FPGA, SmartNICs have proven to be more power efficient than their equivalent on the host.

3.3.1 Kernel Interposition

Even though Kernel Bypass allows drastic gain by avoiding the kernel overhead and data copy [23][4], it has its drawbacks. One of them is the lack of interposition [23].

As kernel bypass allows applications to take direct control of the network interface, we instantly lose any possibility to monitor and control what is happening on the data plane. This happens to be an important problem for network engineering on the edge, as many tools and techniques are now unavailable :

- **Debugging**: When discovering an unexpected behaviour within an application, without kernel bypass, one would use kernel tools and features (*tcpdump*, *ARP tables*, etc.) to understand the source of the problem. Once we went down the bypass road, all the network logic is contained within the application, making all classical tools unusable.
- **Iptables** have been a very used tool for tuning the network behaviour and defining access authorizations and security within the Linux kernel. Once again, bypassing the kernel implies bypassing those iptables.
- **Process scheduling**: The Linux kernel provides useful scheduling for blocking I/O operations, allowing an application to sleep while waiting for an I/O and to be wakened up by the kernel when such I/O occurs. In a bypass situation, blocking calls are not an option, as the kernel cannot detect entering packets. Bypassing applications are thus forced to use polling techniques, therefore burning CPU cycles.

On-Path interposition

There is therefore a need to re-create a form of interposition without losing the advantages of Kernel Bypass. A solution would be to embed this interposition mechanism on a fully programmable *SmartNIC*. Complete integration of these *SmartNICs* within the operating system is possible [23]. It allows an application to notify the OS of a connection in a *socket-like* manner. The OS can then load code on the NIC that can monitor, manipulate and filter traffic in the data plane. Some properties can be expected from this integration :

- **Isolation**: Keeps application from evading policies enforced by the interposition layer.
- **Cross-application traffic *interpositionability***: The interposition layer should provide a global view of the traffic.
- **OS integration**: Interposition must take advantage of its proximity with the host to access the knowledge of the running processes (privileges, how to interrupt them, etc.). This allows kernel interposition to stand out from other in-network solutions such as P4.

- **Avoid unneeded data copies**: One of the principal justifications for kernel bypass is to avoid the data copy bottleneck of the kernel. Interposition must therefore avoid simply moving this problem to the NIC.
- **Fully programmable**: To allow network functions to keep up with the high update rate of software applications, a fully programmable *Smart-NIC* should be preferred over fixed-functions hardware accelerators.

3.3.2 Offloaded Load Balancers

Some initiatives already implement an L4/L7 load balancer[5] on the NIC to dispatch connections across different servers, typically running on different VMs on the host. To do so, the NIC load balancer (NIC-LB) will handle connection establishment from clients, and then establish a connection with the server of its choice according to its load-balancing logic. Once these connections are set up, NIC-LB can relay packets from the client to the server, modifying the payload on the fly (ex: adding HTTP headers).

Modifying the payload on the fly requires carefully adjusting sequence numbers between the two connections, as inserting information changes the byte count.

Additionally, NIC-LB faces some challenges concerning the synchronization of the load balancer's data structures. RSS is complex to use in this situation due to the different addresses of the egress and ingress parts of the same traffic. Instead, a Cuckoo hash table is used to enable a concurrent connection table design.

3.3.3 Netronome NFP NICs

Previous attempts at providing a general networking offload with Linux have encountered certain obstacles [13]:

- **Limited hardware capability**: Trading flexibility for performances is very common, therefore making the provided offload usually stateless and specific.
- **CPU architectures scaling**: General purpose CPU adapted well to network requirements, reducing the interest of a hardware offload with a dedicated processor.
- **Vendor Specific solutions**: Proposed solutions are usually vendor specific, therefore reducing the generalization of offload.

NFP NICs [13] are trying to provide network offload through the general purpose language that is eBPF, in an attempt to circumvent the impediments described previously.

3.3.4 Nvidia Bluefield

Nvidia's Bluefield-2 matches these characteristics. It is an off-path SmartNIC shipping a multicore ARM processor to run a customized Linux distribution. Due to important hardware acceleration, the Bluefield-2 benefits from low CPU costs when processing the host's traffic, thanks to an ASAP^2 [17] embedded switch.

Offloading possibilities

The Nvidia Bluefield is almost systematically less performing than any regular host [15], matching current standards for a data center server. This is expected, as running a SmartNIC with highly powerful hardware (e.g., X86_64 CPU instead of ARM) would likely rapidly exceed the cost of simply adding more CPUs directly in the host. However, it illustrates how solutions built for the Bluefield-2 would require tight integration with SmartNIC hardware accelerators to bring a gain from offloading workloads to the NIC.

3.4 Flow management

3.4.1 Mice and Elephants

Flows tend to be similarly distributed on the internet. While most flows are short in terms of size and lifetime (called mice flows), most of the traffic (~80%) is carried by long flows (called elephant flows), which amounts to a very small proportion of all the connections [8].

A preferential treatment is necessary to guarantee a short response for short flows, as classical flow control mechanisms (such as TCP) tend to favour long flows. Moreover, when considering per-connection scheduling, one must be aware that picking a flow randomly would most likely result in picking a short flow [8]. Identifying a connection as either short or long is of paramount importance.

TCP contextualization

When short and long flows are competing on the same network, some TCP mechanisms tend to favour short flows :

- TCP slow-start phase implies starting a connection with the smallest congestion window possible.
- Short flows don't have enough packets to take advantage of the *dup-ack* mechanism for congestion control due to the limited congestion window size.

A short flow will therefore most likely have to wait for a timeout before detecting a congestion.

- Short flows cannot take advantage of the TCP RTO adjustment mechanism, as it requires a few packets to probe a correct RTO. Thus, short flows will work most of the time with the default RTO value, which is by definition very conservative.

With that in mind, trivial drop-tail queues tend to favour long flows, as they will detect losses more quickly and adjust in consequence. Instead, Random Early Detection-based schemes tend to achieve better fairness for short flows, as they will most likely discard packets from long flows from their queues.

3.5 Virtual Switching

3.5.1 Open vSwitch

Open vSwitch [20] is built as a general-purpose virtual switch, contrasting with many static and hard-coded forwarding pipelines used to provide connectivity to virtual machines. However, this general purpose characteristic leads to important divergences in the switch design :

- Most traditional appliances targeted line rate in worst-case scenarios. This is complicated for OVS, as his general characteristic brings some overhead. Therefore, it focuses on the common case over the worst-case, heavily relying on caching to reduce CPU usage and increase forwarding rates.
- The localization of OVS on the network edges requires scalability as a virtual switch can be connected to thousands of peers, handling VMs state information (Boot, migration, etc.).
- OVS supports OpenFlow, allowing easy switch programming.

Design

An OVS switch is decomposed in multiple components, illustrated in figure 3.3 :

- **ovs-vswitchd**: An operating system independent user space daemon determining how packets should be handled and instructing the *Kernel Datapath* to cache the result, allowing the kernel datapath to process the following packets on its own.

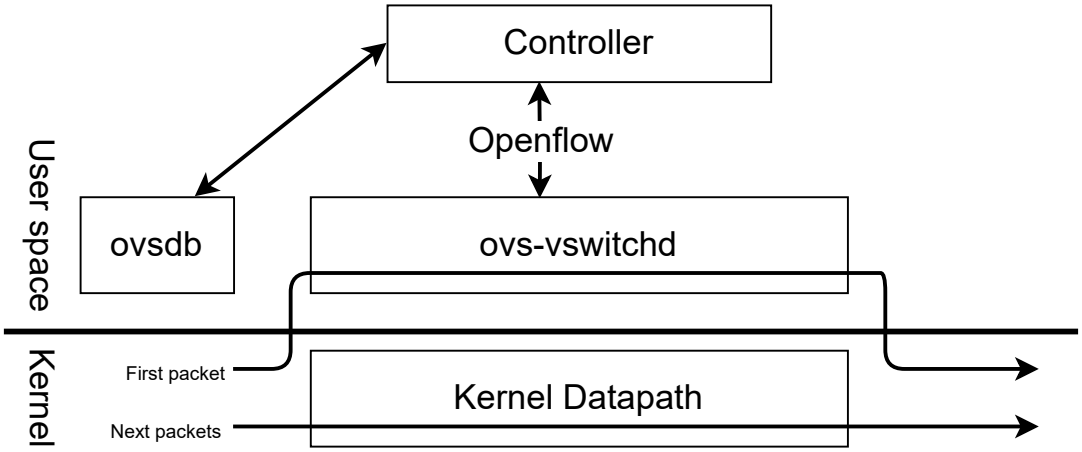


Figure 3.3: Basic components of an OVS switch

- **Kernel Datapath** : OS specific component that receives packets. If *ovs-vswitchd* has previously provisioned actions for the received packet, it will simply follow these instructions. If not, it delivers the packet to *ovs-vswitchd* and follows the steps mentioned above.
- **Controller**: Adds, removes, updates, monitors and obtains flow statistics on flow tables through OpenFlow.
- **ovsdb**: Contains information about OpenFlow switches, their ports, QoS configurations, OpenFlow controller and switches associations, etc.

Flow caching in the Datapath is performed with a two-layer cache [20]: a *microflow cache* that is caching per-connection forwarding decisions and a *megaflow cache* that caches aggregated flow decisions beyond individual connections. To reduce the size of the flow table, OVS proposes a multiple table implementation with an action called *resubmit*. Allowing packets to consult multiple flow tables.

Additionally, OVS extends OpenFlow [16][20] to include meta-data fields called *registers* that can be matched by flow tables. These registers can be used to memorize temporary information at some point in the processing of a packet, and the later process can match this information to select particular actions.

3.5.2 OpenFlow

OpenFlow [16] is born out of a compromise with our current network situation, between switch vendors preferring closed platforms (and consequently providing a low customization level on their hardware) and completely programmable solutions

that fail to satisfy network requirements in terms of throughput and port density. It provides an open protocol to program flow tables in switches and routers, therefore bringing programmability within closed-source and opaque network devices.

Design

An OpenFlow switch is characterized by :

1. *A Flow-Table*: A match+action table allowing the switch to match particular headers of a packet to an action.
2. *A Controller* to which the switch is connected to.
3. *The OpenFlow protocol* structuring communications between the switch and the controller.

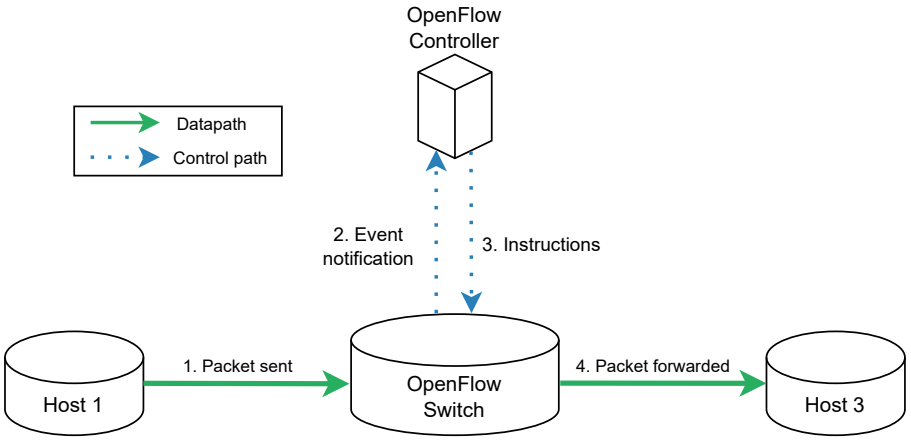


Figure 3.4: Classical representation of an OpenFlow-enabled switch

Behavior

When an OpenFlow switch receives a packet, it tries to match it with its Flow Table to define what action should be taken. Whenever necessary (typically on the first packet of a flow, where there is no matching entry in the Flow Table yet), the switch can encapsulate the packet into an OpenFlow message and send it to the controller. This controller, implementing a user-defined logic, will decide what action the switch should take for this packet by adding a matching entry with a particular action. Once this entry is installed on the switch, every subsequent packet will directly match it and avoid a detour through the controller.

Gain of OpenFlow

OpenFlow [16] successfully brings flexibility into closed-platform network devices. More precisely, it achieves the following goals :

- **High-performance at low-cost**: OpenFlow Switch Match+action phase is expected to run at line rate while designed to be close to existing hardware, therefore reducing the cost for vendors.
- **Capable to support a broad range of research**: With a flexible design where users can match multiple packet headers, it allows a high degree of customization that can be adaptable to many use cases.
- **Isolate experimental traffic from production**: OpenFlow allows packets to be tagged (VLAN, encapsulations, etc.), therefore making it possible to separate production traffic from experimental one. Controllers can thus apply a new logic for experimental traffic while applying a classic logic for production packets.
- **Consistent with vendors' need for closed platforms**: With OpenFlow, a network vendor only needs to expose a secured connection to allow a controller to connect. All the inner workings of the switch can be hidden from users.

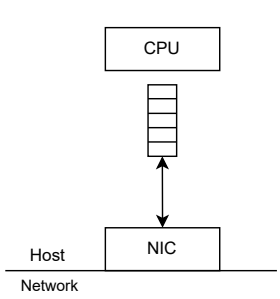
3.6 Packet Scheduling

3.6.1 RSS

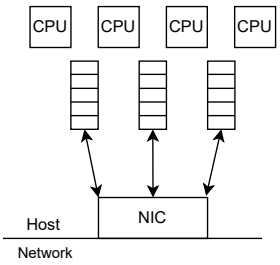
Today's network requirements don't only need higher bandwidth, but also a more efficient processing of data. With the growth of server virtualization, real-time services such as video on demand, and a continuously increasing network bandwidth [10], this need for efficient CPU processing will keep growing.
Without RSS [10], application data requests were associated with a single processor and handled sequentially (figure 3.5a), leading to a high CPU utilization level and difficulty to handle heavy network loads. This configuration tends to cause a bottleneck when working with high bandwidth, reducing the advantage of such a bandwidth.

What RSS brings

RSS implements a table allowing entries to be matched to a particular packet queue. When a packet is received, a hash is computed from certain fields of the packet headers, which can be specified by users. This hash is then matched with the table



(a) Without RSS : Incoming packets are handled by a single core.



(b) With RSS: Packets are dispatched across multiple queues, opening the door for multi-core sharding or lock-based systems.

Figure 3.5: Comparison of packet dispatching with and without RSS. We can see that RSS takes advantage of multicore architectures, maximizing the available bandwidth.

to determine which queue the packet should be sent to (figure 3.5b). This allows packets to be dispatched to different CPU cores or queues to overcome the limit described above.

3.6.2 Elastic RSS

eRSS [22] is an attempt at providing a dynamic core scheduling based on RSS logic. It splits host cores into different *scaling groups*. As their name suggests, these scaling groups can scale independently, with their own queuing logic. Instead of running a balancing mechanism on the host, it takes advantage of a programmable NIC to offload this balancing process.

On the NIC

A program running on a programmable NIC handles the balancing problem. It matches incoming packets 5-tuples to a particular scaling group and then assigns the packet to a core of the associated scaling group. The NIC also keeps estimations of queue depth and tries to balance connections among them by updating the weights used for core assignation.

On the Host

The Host cores have to gather information for the NIC, such as the queues depth or a throughput estimation, that are frequently synced with the NIC. The host must also run a software manager that will allocate cores to the scaling group. The

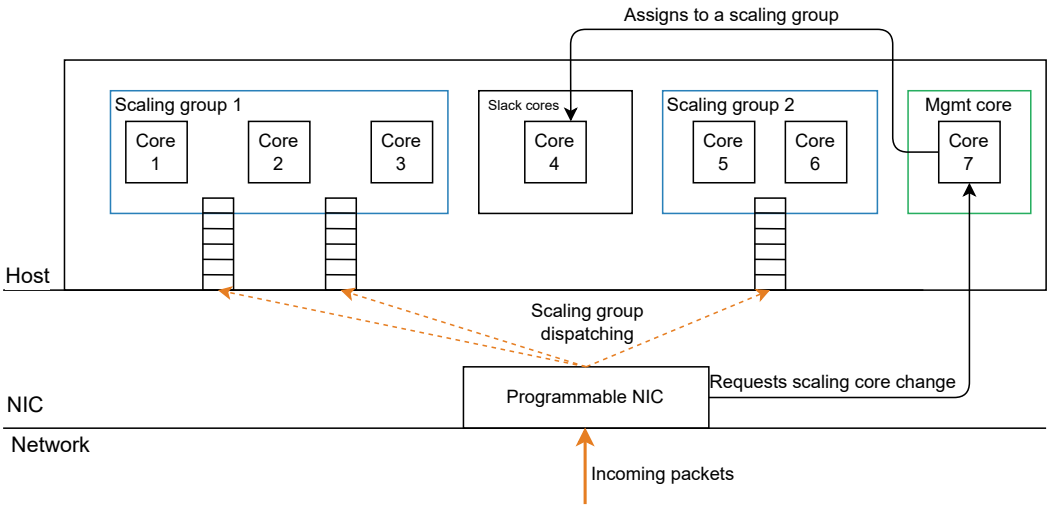


Figure 3.6: eRSS processing pipeline: A programmable NIC decides whether a scaling group should be enlarged, and notifies its decision to a dedicated management core that will allocate slack cores in consequence.

same core cannot be in two different scaling groups at the same time, but a scaling group can claim an unused core from another group.

Difference with oRSS

The important differences between eRSS and oRSS are the refinement level of the scaling and the CPU consumption on the host. eRSS queue-based load approximation keeps it from applying a more refined scaling, while oRSS per-flow load approximation allows such fine-grain scaling. Moreover, eRSS requires a dedicated core on the host, while oRSS dedicates all its running cores to data processing. These divergences are due to a slightly different objective for both oRSS and eRSS, the first attempts to dispatch flows adequately among cores while the latter is focused on using the least cores possible for the application as a whole.

3.6.3 FlowValve

Offloading packet scheduling on a SmartNIC can be an answer to inefficient software schedulers nibbling applications' CPU cycles and inflexible on-NIC schedulers keeping users from defining a complex scheduling policy [28].

FlowValve [28] allows users to define QoS policies that will be transformed into a scheduling tree that describes class relationships and bandwidth distribution policies. This tree is used to populate filter rules into an NFP SmartNIC. This

allows FlowValve to enforce more complex policies than some Linux Traffic Control such as *Hierarchy Token Bucket* while reaching speeds as high as 40Gbps [28].

Offloading efficiency

While other solutions to enforce similar QoS policies in the host exist such as a DPDK QoS scheduler and can enforce these policies, this comes at the cost of extensive host CPU consumption. Meanwhile, *FlowValve* archives the same objective without such host CPU utilization [28].

3.6.4 RSS++

Multiple software architectures leverage multiple CPU cores to serve multiple requests in parallel, aiming to meet high demands with low latency [1]. In such a configuration, *sharding* can be useful as it divides resources into multiple completely independent shards [1]. Each of these shards usually maintains a space allocated for packets sharing the same characteristics. In a network workload, this poses the problem of balancing flows. Without proper balancing, some CPU cores might quickly become overloaded while other cores are waiting, thus leading to a waste of resources. More precisely, sharding packets over different cores requires the following :

- **Load-balancing** to avoid overloading certain cores while others are idle.
- **Flow-to-core affinity**: Packets from the same flow must be served by the same shard, thus eliminating directly the possibility of sharding solely based on the buffer utilization.
- **Migration minimization**: The number of flow-states transferred between shards to balance the load must be minimized, as it leads to more overhead.

There is software that allows this sharding, such as RSS [10]. RSS++ is an attempt to implement a mechanism of load balancing within RSS. This implies moving flows between cores to ensure that loads are spread around cores as much as possible. This mechanism is implemented through counters that are maintained by each core. Regularly, RSS++ will gather these counters to detect any under/over-used core, and balance load accordingly. In practice, the load balancing is implemented with a Greedy method, trying to move the most overloaded bucket of the most overloaded core to the least loaded core repeatedly until core loads are within a 1% margin of the average value.

Improvement compared to RSS

RSS++ improves RSS throughput by up to 10%, while significantly reducing both the latency and the number of dropped packets. This illustrates the effects of load balancing: avoiding a too-long waiting time in a crowded queue and therefore avoiding full queues that must drop packets. In terms of CPU, *RSS++* dynamic CPU cores utilization allows using only the required amount of cores, while RSS uses a fixed number of cores. Coupled with the other mechanisms of *RSS++*, this leads to a drastic improvement, with more than 3 times fewer CPU cores required by *RSS++* on the same workload. Additionally, *RSS++* provides a lower imbalance and tail latency.

3.6.5 Programmable Packet Scheduling

Packet scheduling in switches is usually non-programmable, as operators have to choose among scheduling algorithms implemented by the manufacturer [24]. Historically, network switches have been fixed-function devices, implying a large delay to deploy new features as they request a hardware re-design. Even if some new *programmable* switches allow operators to specify a new protocol format, the packet scheduler itself remains hardwired [24].

Decomposing scheduling

Any scheduling algorithm can be thought of as two different decisions: *when* and *in which order* should packets leave the switch [24]. *PFabric*, *Weighted Fair Queuing* and *token bucket traffic shaping* are so many algorithms that only differ by the order and time of the leaving packets. This leads to an abstraction built around two types of queues: **Priority queues** and **Calendar queues**. Complex scheduling algorithms such as *Hierarchical packet-fair queuing* can use multiple priority or calendar queues combined, but are still expressible through these fundamental blocks.

Hardware feasibility

Both the calendar and priority queues can be implemented in hardware with the same building block: A *Push-In First Out* queue (a queue where items can be en-queued at any location, but are dequeued following the head of the queue). At the en-queue time, a packet position in the queue can be determined directly by its headers or from a persistent state maintained within the switch (Wall-clock departure time, etc.). This PIFO design revolves around a hardware-sorted queue, which is already available in some hardware. A programmable packet scheduling is already doable in the hardware [24], even if quite limited in terms of capacity.

3.6.6 Informed Request Scheduling

RSS works well in a situation where workloads are fairly equivalent [9][1]. Highly-variable execution times have proven to be a more challenging task, requiring scheduling across cores and preemption. Current solutions tend to dedicate a CPU core to achieve these scheduling tasks, but they scale poorly. More precisely, they usually suffer from either load imbalance, lack of preemption, limited scalability or inter-thread overhead. Instead of choosing between blind cheap scheduling at the NIC and costly informed scheduling on the host, a solution could be to perform informed scheduling directly on a SmartNIC. This would require the SmartNIC to fill some properties [9] :

1. The SmartNIC should be able to address requests to specific cores.
2. The host server must be able to communicate with the SmartNIC in fine granularity.
3. The SmartNIC must be aware of system and queue states
4. The SmartNIC must be able to interrupt server cores to implement preemptive scheduling.

This paper[9] shows that it is possible to improve current scheduling through SmartNIC offloading, as a SmartNIC has a centralized view of all packets entering the system.

Chapter 4

Design

4.1 Overview

This work aims at providing a SmartNIC offloaded packet scheduler. As described in the previous section, RSS is performing well when handling flows of similar weight. When processing mixed-type of flows (e.g., multiple CPU-light and few CPU-heavy flows), the probability for RSS to hash multiple CPU-heavy flows on the same core increases, and generates packet losses and delays. RSS++ tries to counter-balance this problem by dynamically re-balancing flow buckets between cores to avoid high workload imbalances while preserving flow-to-core affinity. The downside of this solution is its additional CPU consumption to perform balancing, consequently reducing the number of cycles available for data processing. This work presents *oRSS* (offloaded RSS), a solution that mimics RSS++ behaviour while taking advantage of SmartNIC offloading to relieve end-host cores from as much control logic as possible while ensuring that packets processing's load is spread fairly across cores. To do so, oRSS orchestrates flow migrations between cores, allowing a flow to move from one core to another.

While oRSS is presented as an integrated solution, it is actually the result of cooperation between two distinct implementations :

- **oRSS-Host**: A DPDK [14] application running on the end-host, therefore consuming important CPU cycles. It is required to gather information about the usage of each core. Obviously, this work aims at making this implementation as light as possible to reduce its CPU consumption.
- **oRSS-NIC**: A classical application running on the SmartNIC, and is therefore completely independent of host CPU. It aims at gathering usage information from oRSS-Host to decide whether flows must be re-balanced or not.

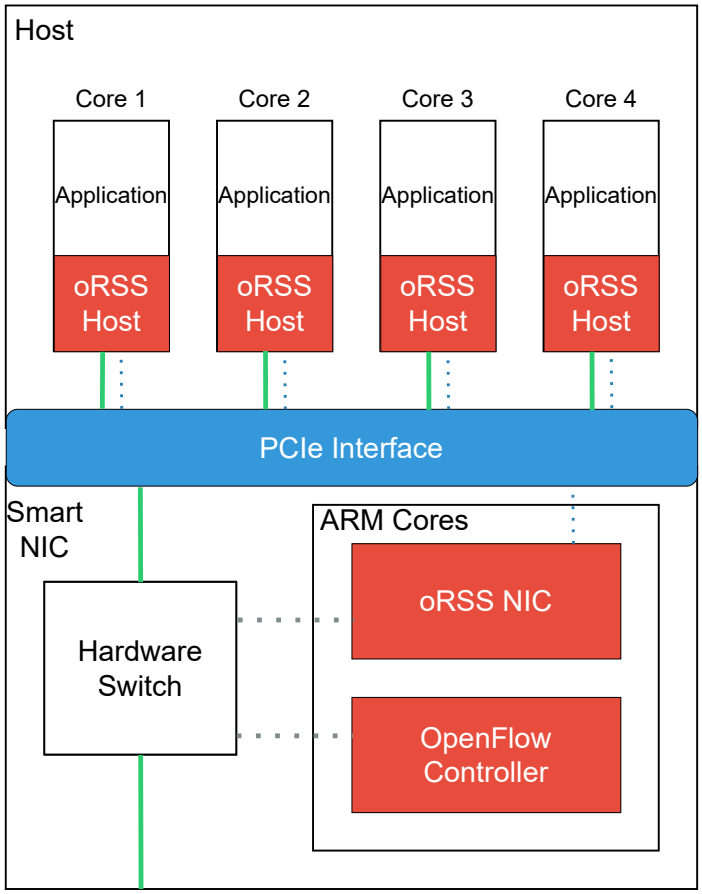


Figure 4.1: oRSS: Architecture overview

This architecture is detailed in figure 4.1 and the source code is available [25]. In the remainder of this section, we'll see in detail how each piece of oRSS is implemented and how they cooperate in order to perform an efficient flow re-balancing logic.

4.2 Host side

The oRSS-Host program embeds a very simple logic, relatively close to what a classical RSS-based program would look like. Figure 4.2 presents an activity diagram showing oRSS-Host logic. It is built around an infinite loop that keeps polling for packets. Whenever the host receives enough packets to fill a burst, the function `rte_eth_rx_burst()` will return an integer representing the number of packets available and fill a buffer with those packets.

Then, oRSS-Host will iterate over each packet and parse it. As we'll see in section 4.6, processing each packet as they arrive not be the most efficient thing to do in case of ongoing migrations. The program will use its integrated migration logic to determine whether a packet should be processed or buffered for later processing. In the latter, it will simply store the current packet in a buffer. If the host decides to process the packet, it will store information (i.e., packet 5-tuple) about the current packet in a shared data structure made available to *oRSS-NIC* through a DMA connection.

4.2.1 Sharding

As shown in figure 4.1, oRSS-Host actually consists of multiple instances, each pinned to a particular core. Each of these instances runs independently of others, with their own memory. It thus consists of a shard, without shared memory between cores. Any cooperation between cores (i.e., flow migrations) is performed directly by the NIC. This design allows oRSS-Host to circumvent many of the multicore programming problems (e.g., Slow atomic operations, false sharing, etc.) while offloading cores cooperation to the SmartNIC.

4.2.2 Overhead

As we intend to offload as much as possible of the control logic to a SmartNIC, it is important to identify sources of overhead. Although this design is thought to minimize these overheads, we can still see different origins for these slowdowns to occur compared to a classical RSS program :

- **Checking for migrations and buffering**: This is the main source of overhead in this implementation, it requires the host to iterate over a list of ongoing migrations to see whether a packet should be processed. For efficiency reasons, this iteration is strictly restricted to actual ongoing migrations towards the concerned core. This makes the overhead variable, as it depends on the number of ongoing migrations. But as migrations should allow current flow distribution to find an equilibrium, we can assume that the number of migrations will be fairly reduced once flows are balanced. In the case of buffering, only a pointer towards the parsed packet structure is copied into a buffer.
- **Handling buffered packets**: Similarly to what we described above, this action requires looking over current buffers to check if some can be emptied. Once the host detects that a buffer can be emptied, it simply processes every packet it contains in a row.

- **Updating shared structures**: Before and after each packet processing, the host stores information about the current processed packet into a DMA-shared data structure. As oRSS handles flows on a per-connection basis, this involves copying packet 5-tuple (i.e., IP source/destination, port source/destination and IP protocol field) before packet processing and simply flipping a validity bit to 0 afterwards. Additionally, each running instance of oRSS-Host keeps a packet counter that is also shared with the NIC.

4.3 NIC side

The NIC side is way less homogeneous than the host. It consists of multiple components serving different purposes.

4.3.1 Hardware switch

An essential element to understand is that packets don't flood through SmartNIC's ARM cores. Due to ARM's weak performances, such a design would rapidly introduce an important bottleneck that would cap Bluefield performances below what it can physically provide [9]. Instead, we take advantage of an NVIDIA ASAP² [17] programmable switch. This technology allows a hardware switch to be controlled through software running on the ARM cores. In the case of oRSS, programs on ARM cores communicate with the switch through an offloaded OVS [20] [18] instance. More precisely, both oRSS-NIC and the OpenFlow Controller visible in figure 4.1 connect to the OVS instance with the OpenFlow protocol to control the OVS switch and thereupon to control the hardware switch.

4.3.2 OpenFlow Controller

This is an easy-to-use Python controller built with the Ryu framework [7]. It is meant as a customizable element to allow users to define their own logic within oRSS. Users can thus specify conditions under which packets should be dropped, forwarded, mirrored, etc. It is also the component that handles fresh connections. Whenever a new connection occurs, its first packet is transmitted to this controller, which by default inserts a new rule into the switch that forwards the following packets to the host and wraps them into two VLAN tags [26]. The first VLAN indicates to the host to which core a packet should be delivered while the second provides a unique identifier to identify the flow without parsing the complete payload. When first inserted, the VLAN indicating the destination core is chosen randomly. This allows oRSS to act as classical RSS at first before flows are balanced as described in section 4.5.

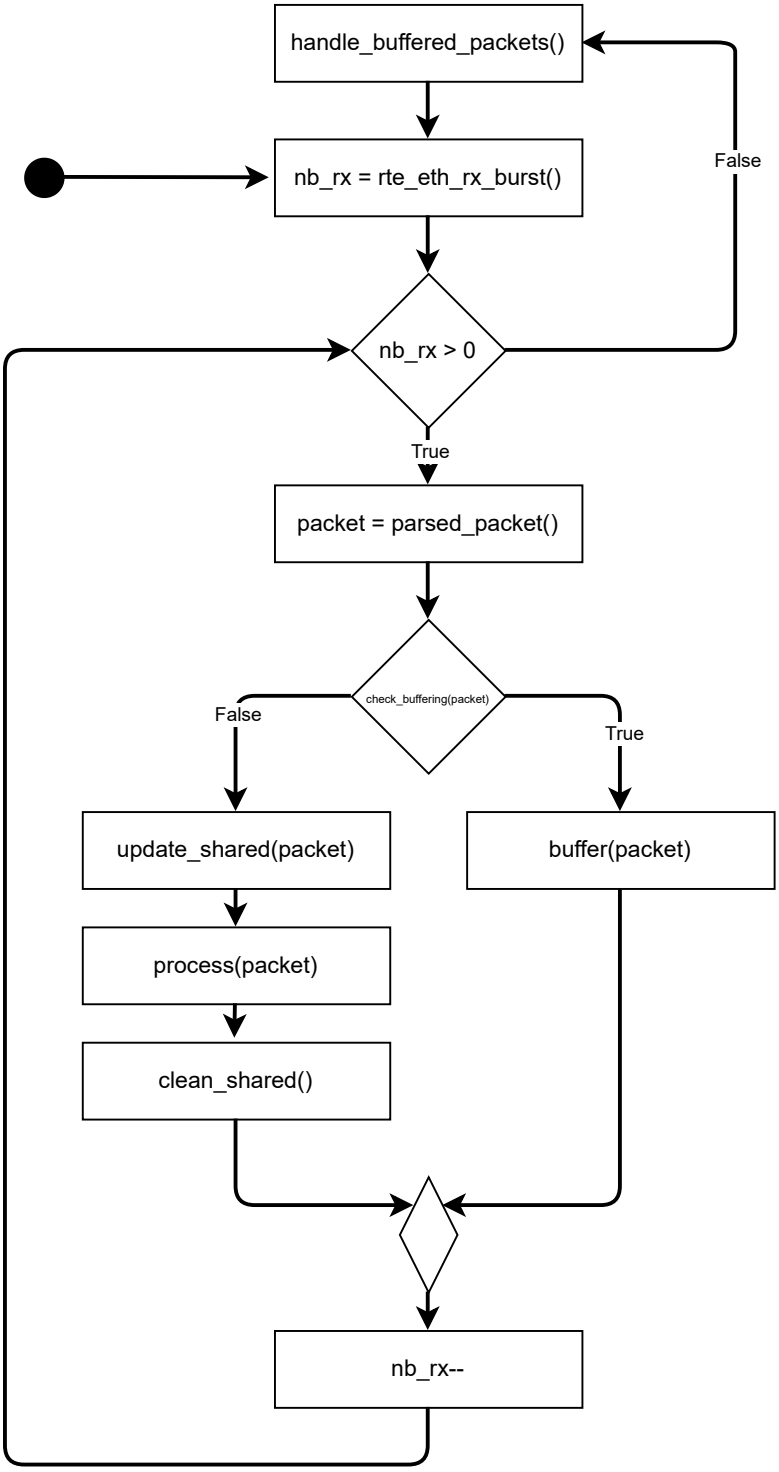


Figure 4.2: oRSS-Host: Activity diagram

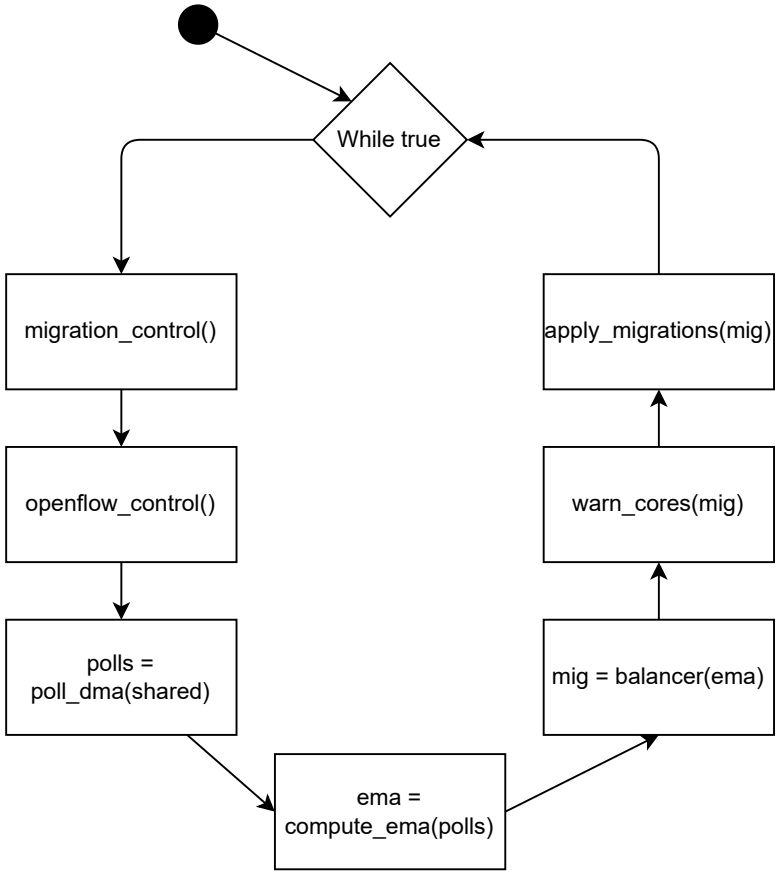


Figure 4.3: oRSS-NIC: Activity diagram

4.3.3 oRSS-NIC

It is certainly the core element of oRSS. It handles all the balancing logic and core usage computation. Through DMA connections, it can write and read data directly into each oRSS-Host instance memory. Through this mechanism, it polls each core frequently to estimate its usage. It is also connected to the Hardware switch through an OpenFlow connection. Figure 4.3 unveils its inner working. We'll now go into details of each of these fundamental pieces :

- `migration_control()` : Checks every ongoing migrations and perform required actions described in section 4.6.
- `openflow_control()`: If there weren't any communication between oRSS-NIC and the OVS switch for a while, OpenFlow protocol requires the controller (in this case, oRSS-NIC) to respond to a PING message. This function looks for incoming messages from the switch and responds to it if necessary.

- `poll_dma(shared)`: Performs the polling of shared memory described in section 4.4, to gather the required information in order to approximate the load of each core.
- `compute_ema(polls)` : Computes the exponentially moving average as showed in section 4.4. It approximates the CPU usage of each flow to detect congestion.
- `balancer(ema)` : Main function of the whole oRSS project. It takes the computed EMA averages and decides whether flows should migrate to another core. It outputs a list of migrations.
- `warn_cores(mig)`: Before actually applying migrations, oRSS-NIC warns each core that will receive a connection. This information is used by the oRSS-Host to decide whether a packet should be buffered.
- `apply_migrations(mig)`: Now that cores are warned about their incoming migrations, oRSS-NIC can apply them. To do so, it sends a `FLOW_MOD` [27] OpenFlow message to the switch to change rules installed by the OpenFlow Controller. More precisely, it will change the VLAN associated with the rule.

We can thus see that oRSS-NIC is the key element that makes oRSS balanced, to counter the static property of classical RSS.

4.4 Core load approximation

Now that we saw the details of each component or oRSS, we can dive into how they coordinate to achieve balancing. The first and most important communication between oRSS-Host and oRSS-NIC concerns core load exchange. To balance flows, oRSS-NIC must be aware of per-flows CPU consumption on the host.

4.4.1 Clock-based core load

A first solution would be to let oRSS-Host measure CPU cycles for each packet it processes and store this information in a shared data structure. While this implementation has the advantage of providing a very precise and detailed measure of CPU consumption, it comes at an important cost to the CPU itself, as calling the clock frequently turns out to be costly in terms of CPU cycles. Moreover, the oRSS-Host must maintain a dynamic list of current flows and their load. Thus, this measurement mechanism adds significant overhead on the host's side, which can be counter-productive as oRSS aims at leveraging offloading to actually free CPU cycles for data processing.

4.4.2 Polling-based core load

Another approach is to use a single and very simple shared data structure where oRSS-Host stores currently processed flow. Frequently, oRSS-NIC will have to poll this shared structure to identify which flow is occupying the host core. Once performed multiple times, this information can be gathered to estimate per-flow CPU consumption. This approach has the advantage of a very limited cost on the host's side, as it limits itself to copying flows ID into a shared structure, but may suffer from approximation. Instead of polling the host once to get exact measurements, the NIC now has to perform multiple polling in order to approximate a core usage, introducing delays between balancing.

Consequently, we can deduce that this polling method requires some kind of equilibrium between low latency and accuracy. The first is important to detect core congestion as early as possible while the latter requires loads of polling between two rebalancing in order to get a true image of CPU consumption.

Exponentially Weighted Moving Average

A solution to this dilemma is to take advantage of an *Exponentially Weighted Moving Average*. As described in section 4.3.3, oRSS-NIC consists of an infinite loop that polls the host at each iteration. Instead of performing loads of polling at each iteration, oRSS only performs a limited number of polling each time. It combines this information with previous polling to build an estimation of processor consumption. But using previous polling exposes oRSS to the following problems :

- **Outdated information**: Polling from the previous iteration is obviously older, and may not be accurate anymore (i.e., some flows might not exist anymore, some flows might weigh differently now, etc.).
- **Previous balancing**: oRSS-NIC can move flows across cores at each iteration. Thus, older polling might be inaccurate as they reflect a configuration that possibly doesn't exist.

We can thus affirm that the older a polling is, the more likely it is to be inaccurate. With that fact in mind, the usage of an *Exponentially Weighted Moving Average* makes sense. In oRSS, that consists of a ring buffer that keeps track of previously polled values, as described in figure 4.4. At each iteration, the NIC will only fill a fraction of this ring buffer with recent values. Then, it will compute the core usage by giving a certain weight to these recent values, and less to older ones. More precisely, the weight of polling decreases exponentially with their age. This is illustrated in figure 4.5. These weights are customizable depending

on the required utilization. Putting more weight on the most recent polling will reduce the importance of previous ones. Therefore, this causes a more unstable yet reactive average while diminishing the weight of recent polling to produce a more conservative average.

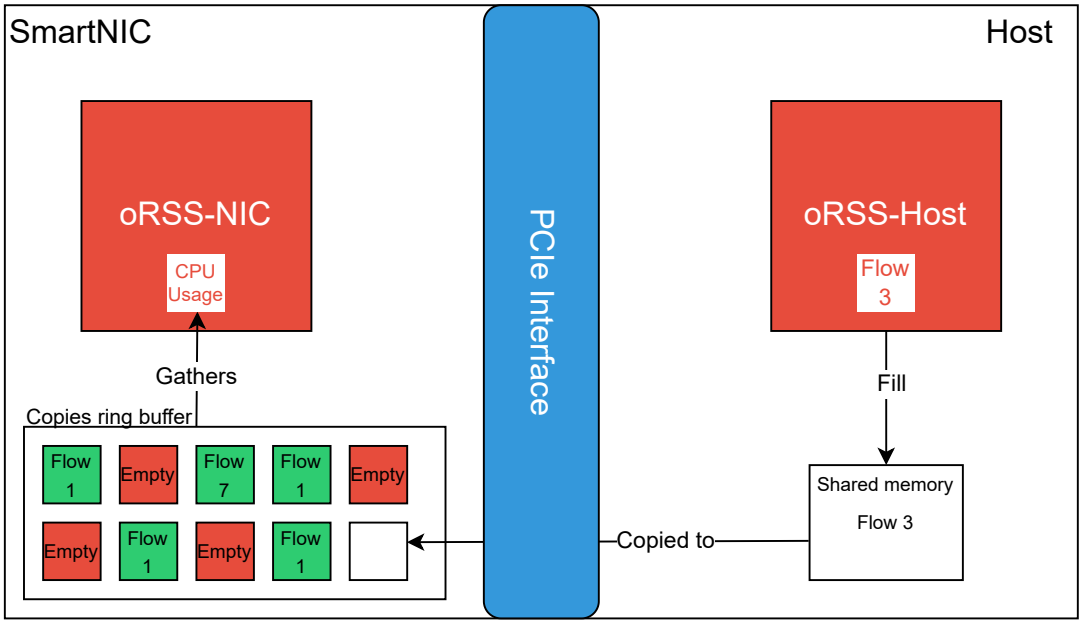


Figure 4.4: oRSS-NIC keeps polling the host to obtain a meaningful view of CPU consumption. These copies can be gathered to compute an estimation of each core utilization.

4.5 Flow balancing

Since the beginning of this chapter, we saw that flows are originally randomly distributed by an OpenFlow controller (section 4.3.2) and how oRSS-NIC measures host CPU consumption. Exactly like classical RSS, even if this random initial distribution ensures the allocation of a similar amount of flows on each core, it doesn't take into account the individual consumption of each flow. While allocating flows randomly, the controller could allow 2 computation-heavy flows on a core and 2 computation-like flow on another core, possibly causing packet drops and increasing latency due to filled queues.

This is where *oRSS-NIC* makes sense. With per-flow core usage gathered as described in section 4.4, the NIC is able to detect when overload happens and

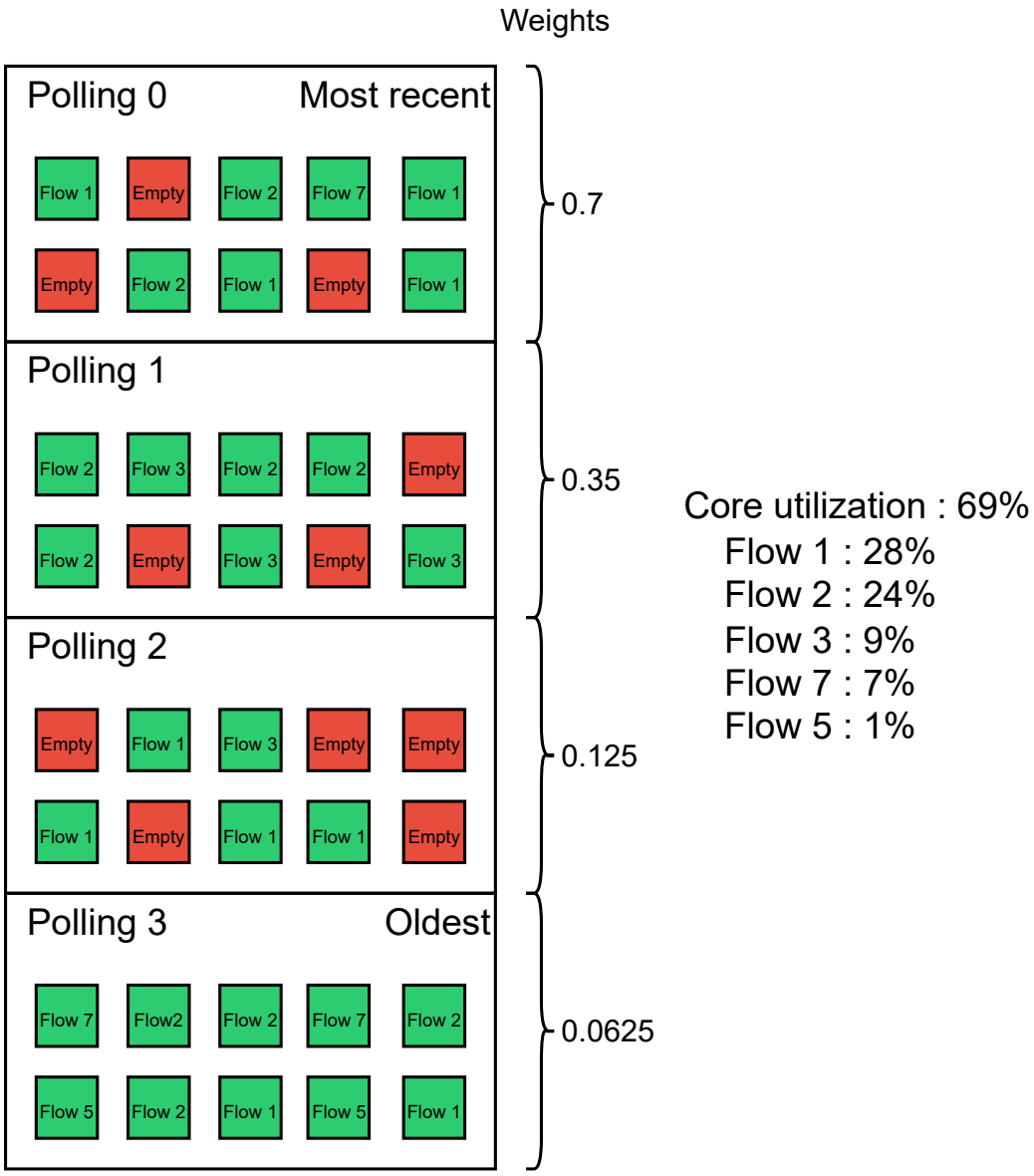


Figure 4.5: oRSS polling example where oRSS-NIC polls the host by batches of 10. Each new polling pushes down the previous ones. When computing core usage, oRSS-NIC gives a certain weight to the most recent polling and diminishes the weight of older polling exponentially. Polling 0 is from the current iteration, polling 1 is 1 iteration old, etc. It is thus capable to approximate the utilization of a CPU core.

whether it is possible to rearrange flows to ensure a fair flow distribution in terms of CPU occupancy. This comes in two steps. First, the NIC has to detect that some flows must be re-dispatched among cores. Secondly, it has to apply these changes while avoiding packet reordering. The first step is described in the remainder of this section, while the latter is discussed in section 4.6.

4.5.1 Load imbalance

In order to rearrange flows among cores to make a better flow distribution, we first need to define what is a better distribution. To do so, we introduce a measure: the *Average Squared Imbalance*. Instead of considering the average core usage, which doesn't take into account the difference in loads between cores, we'll use core usage variance. As illustrated in figure 4.6, this measure indicates how much individual core usage tends to be close to the average usage. For the sake of clarity, we'll refer to the difference between an individual core usage and the average as the *imbalance*. Thus, when considering two flow distributions, we'll consider the one with the smallest average squared imbalance as better. The following section describes the algorithm applied by oRSS-NIC in order to reallocate flows among cores in a way that minimizes the imbalance.

4.5.2 Balancing algorithm

The NIC embeds a balancing algorithm similar to *RSS++*[1]. It uses a user-definable threshold. Once the average squared imbalance (ASI) exceeds this threshold, the algorithm tries to move the largest flow of the most loaded core towards the least loaded core. Then, it recomputes the ASI and sees whether the most and least loaded cores have changed due to this movement. If the flow migration leads to a reduced ASI, then the migration is accepted, and the algorithm repeats the same steps until one of the following conditions is fulfilled :

1. The algorithm reaches a state where moving the largest flow of the most loaded core to the least loaded core doesn't improve ASI.
2. The algorithm reaches the limit of iteration it can perform in a single rebalancing action, which is user-definable.

This logic is illustrated in figure 4.7.

Core reduction

The algorithm as it has just been described suffers from some kind of *head-of-line blocking*. If the largest connection of the most loaded core cannot fit into any other

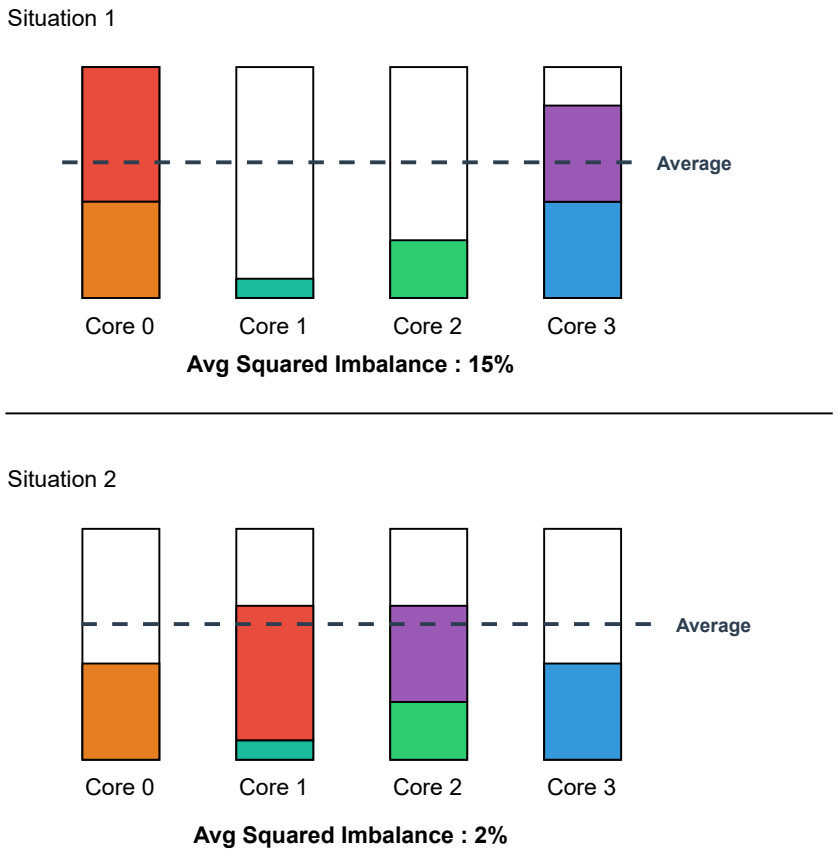


Figure 4.6: Showing how average squared imbalance can characterize the fairness of a certain flow distribution. In the first situation, we can see that cores 0 and 3 are overloaded while cores 1 and 2 are almost idle, leading to a very high average squared imbalance. On the contrary, the second situation exhibits more fairly distributed flows, causing a much lower squared imbalance.

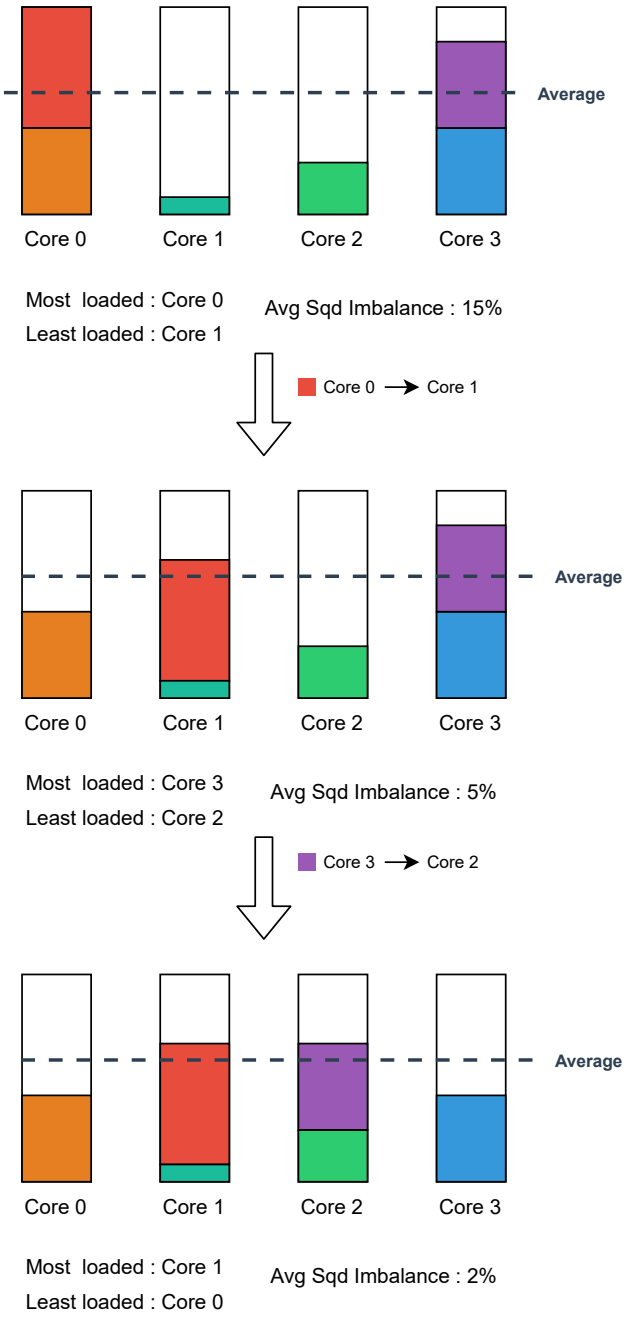


Figure 4.7: A balancing example. In the first iteration, the red flow is moved from core 0 to core 1 to reduce the imbalance. Because of this movement, core 3 becomes the most loaded core and core 2 the least loaded. Reapplying the same logic, the purple flow is then moved from core 3 to core 2. Most and least loaded cores then become respectively core 1 and core 0, but moving core 1's biggest connection to core 2 would actually increase the ASI. The algorithm thus stops on this local optimum.

core, the algorithm would just stop. This is problematic in the presence of a very heavy flow that would occupy most of a core by itself. In such a situation, it is likely that this flow cannot fit in any other core, and would therefore block any migration as long as it exists. To face this problem, the algorithm applies a *core reduction*. It will reduce the number of cores to consider, excluding the most loaded core from the balancing. Then, it can try to reapply the logic described previously to reduce the ASI as much as possible. Figure 4.8 provides an overview of this mechanism.

4.6 Handling migrations

Until now, we have seen how oRSS-NIC gathers CPU load information and how it uses it to decide to migrate flows between cores. The last part of this puzzle is how it actually performs a migration once it is decided. As mentioned before, the NIC applies migration with simple `FLOW_MOD` messages sent to the hardware switch. But it is only the beginning of a whole process.

4.6.1 Packet reordering

Figure 4.9 emphasizes the problem of flow migrations. As migrations intend to move flows from the overloaded to the underloaded core, it is likely that the core receiving the flow has a fairly empty queue while the relieved core has probably a more crowded one. Because of this, fresh packets arriving at the newly assigned core could be processed while older packets are still waiting in the old core's queue. This would force the application to reorder its packets, losing precious CPU cycles for actual data processing.

4.6.2 Reordering prevention

To answer this problem, oRSS divides migrations into two different phases. First, packets are effectively forwarded to their new core. But instead of directly processing them, they are buffered. Once a core starts buffering packets, it signals to the NIC that the migration took effect. The NIC, when polling hosts as described in 4.4, also retrieves a packet counter for each core. With that information, and knowing the constant size of a queue, the NIC can deduce when a relieved core has processed every packet that was in its queue at the time of the migration. Once this happens, oRSS-NIC notifies the destination core that it can safely process these packets. oRSS-Host can now empty its buffer and start processing the newly arrived flow normally.

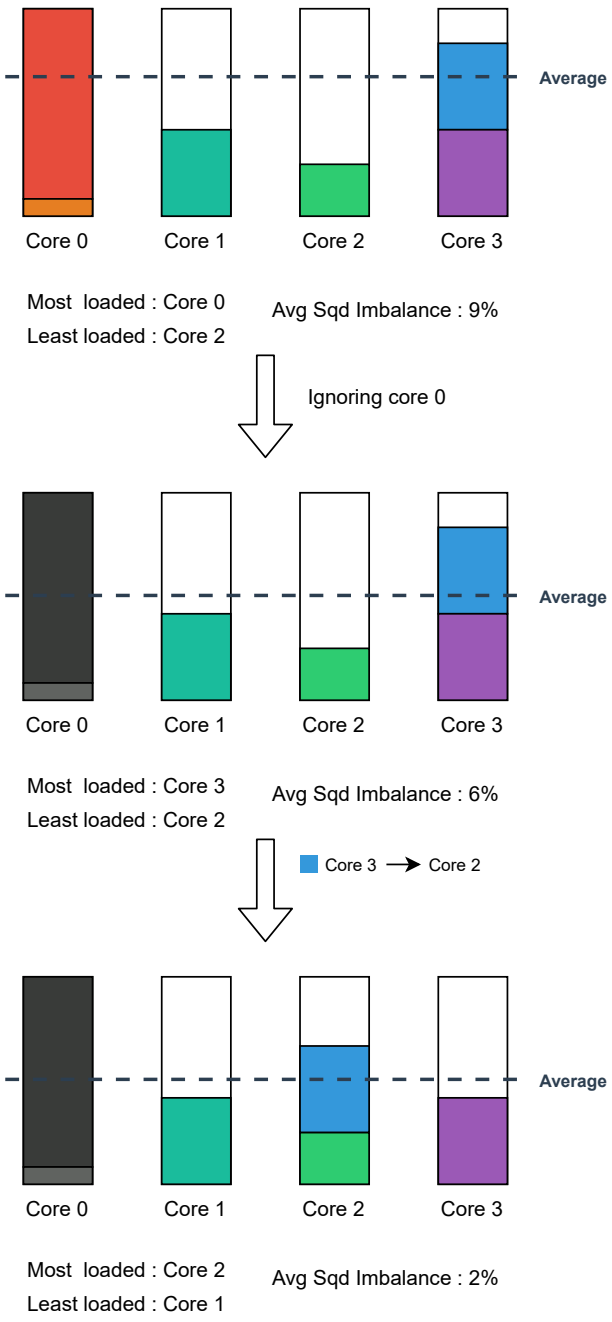


Figure 4.8: In the first iteration, we can see that the red flow keeps us from balancing, as it cannot fit anywhere. The algorithm will thus ignore core 0 and balance the remaining cores.

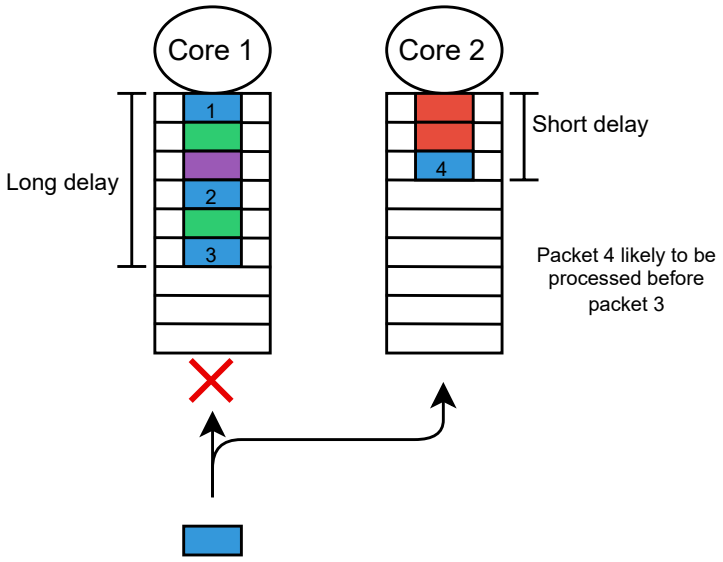


Figure 4.9: Illustration of a reordering situation, where a blue flow is moving from core 1 to core 2. Because of core 1's heavy utilization, its queue is crowded. If done recklessly, the migration might lead to core 2 processing blue packets while core 1 is still processing the flow, causing reordering and breaking the sharding.

4.6.3 Per-flow Datastructures

In order to allow stateful flow processing, a user-defined datastructure is allocated at the first packet of each flow. When a migration is decided, a pointer to this structure, which is a 64 bits integer, is communicated along with migration states described previously. When a core is notified by the NIC of an incoming migration, it also receives this pointer. Thanks to reordering prevention, oRSS ensures that two cores will never attempt to access this shared structure at the same time, therefore relieving the application from implementing any locking mechanism.

Chapter 5

Evaluation

5.1 Testbed

This section provides tests comparing oRSS with classical RSS. Every test described in this section has been conducted with two hosts equipped with DPDK compatible NICs, where a host is running RSS/oRSS while the other is generating traffic with *PktGen-DPDK* (Hereinafter referred to as *Traffic Generator*). These two hosts are running with *Intel Xeon Silver 4314* CPU with 16 cores clocked at 2.40GHz. They both run with Ubuntu 20.04.2 with a 5.13.0-23 Linux kernel and DPDK 21.11.0. On each test, RSS and oRSS are running over a Mellanox Technologies MT42822 BlueField-2 ConnectX-6 SmartNIC. This Bluefield NIC is equipped with 8 aarch64 Cortex-A72 ARM cores running an Ubuntu distribution, shipped with a modified version of the 5.4.0 Linux kernel. These ARM cores communicate with the host through DMA connections handled with *Nvidia DOCA* 1.3.0012. The traffic generator host uses a ConnectX-5 *Mellanox Technologies MT27800 Family* and

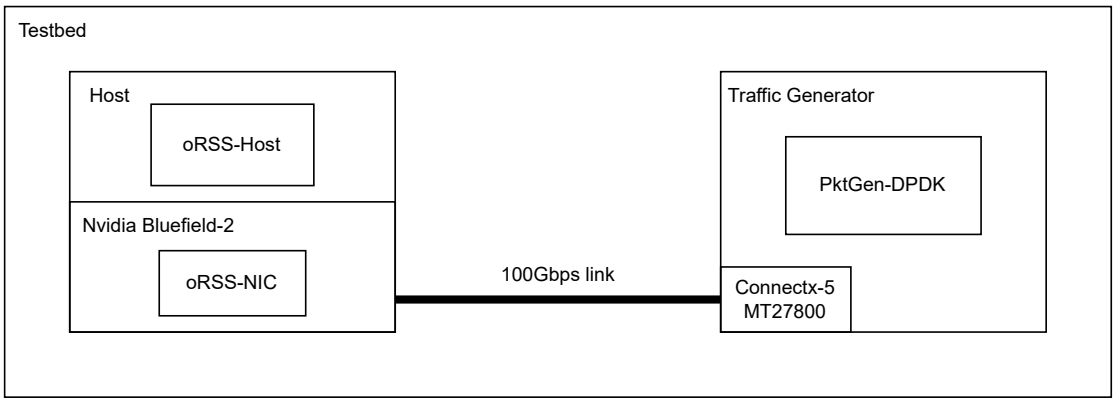


Figure 5.1: High-level representation of our testbed

PktGen-DPDK version 22.04.1 configurable with a Lua script. The two hosts are connected with a 100Gbps Ethernet link. We can define two hardware limitations: The 100Gbps link induces the bitrate limit, while the ASAP^2 programmable switch embedded into the Mellanox Bluefield-2 limits the maximum number of packets to 33M packets per second [17].

For timing reasons, tests are limited to a maximum of 8 cores to run RSS and oRSS-Host. An artificial workload has been tailored to require a complete usage of these 8 cores in order to reach the linerate throughput. Packets have a fixed size of 1500 bytes and generate an equal workload. It thus requires 8.3Mpps to saturate the link. In these tests, oRSS-NIC keeps track of the last 1000 polling for each core and performs 100 new ones at each iteration. It thus computes an EMA over 10 rounds. The weight of round n ($0 \leq n \leq 9$) is determined as :

$$w_n = \frac{0.15}{1.1^n}$$

Therefore, most recent polls are given a weight of 15%, which is fairly conservative. Finally, the migration buffer size is limited to 1024 packets, exceeding packets are dropped but still counted.

5.2 Profiling

In this section, we'll look in detail into profiling of oRSS-Host, oRSS-NIC and classic RSS. To do so, we ran both oRSS and classical RSS on the testbed with 32 flows generated from the traffic generator.

5.2.1 oRSS-Host and RSS

Figure 5.2 illustrates the cost of oRSS in terms of CPU usage on the host. Globally, it produces an overhead of approximately 10%, reducing the time allocated for actual packet processing. This overhead is dispatched through migration processing, retrieving the connection ID at each packet and diverse delays referenced within *Others*. It is worth mentioning that this ID retrieval is induced by hardware limitations to support oRSS current implementation [6]. Indeed, parsing two consecutive VLANs in hardware isn't supported by our driver, forcing oRSS to manually parse it at each packet.

5.2.2 oRSS-NIC

Figure 5.3 shows CPU consumption for oRSS-NIC, running on the SmartNIC. Unsurprisingly, *Polling* and *Usage estimation* use most of the ARM core. If

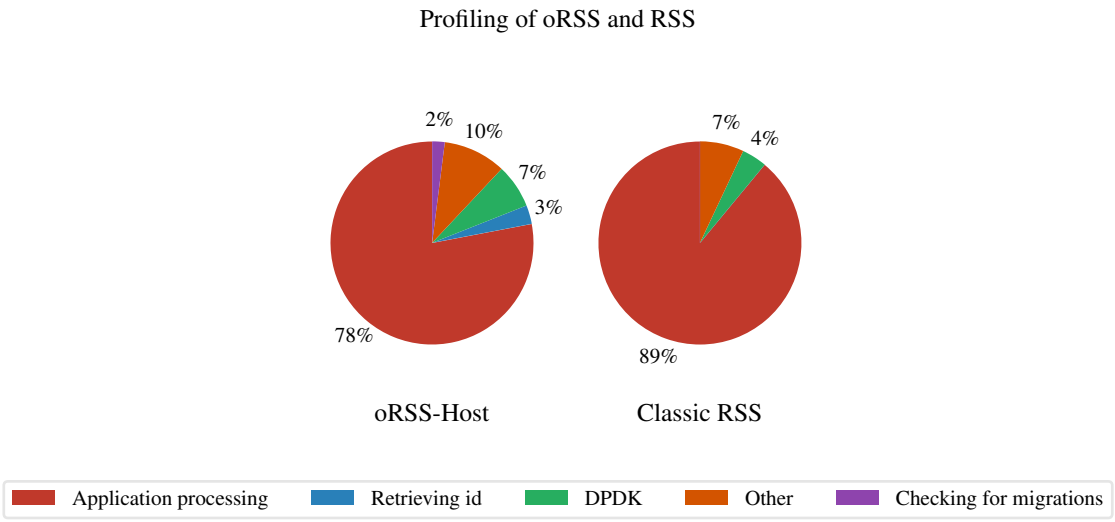


Figure 5.2: CPU profiling for oRSS-Host and RSS with a 32 flows workload

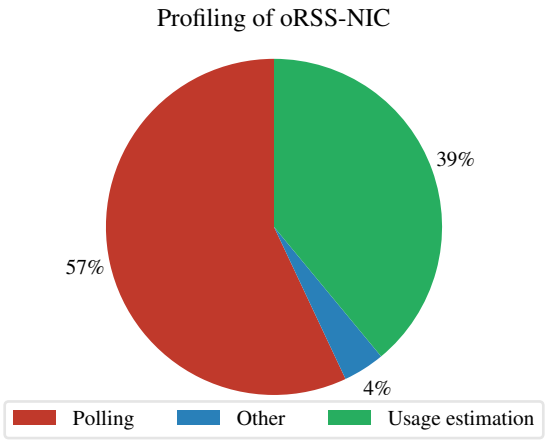
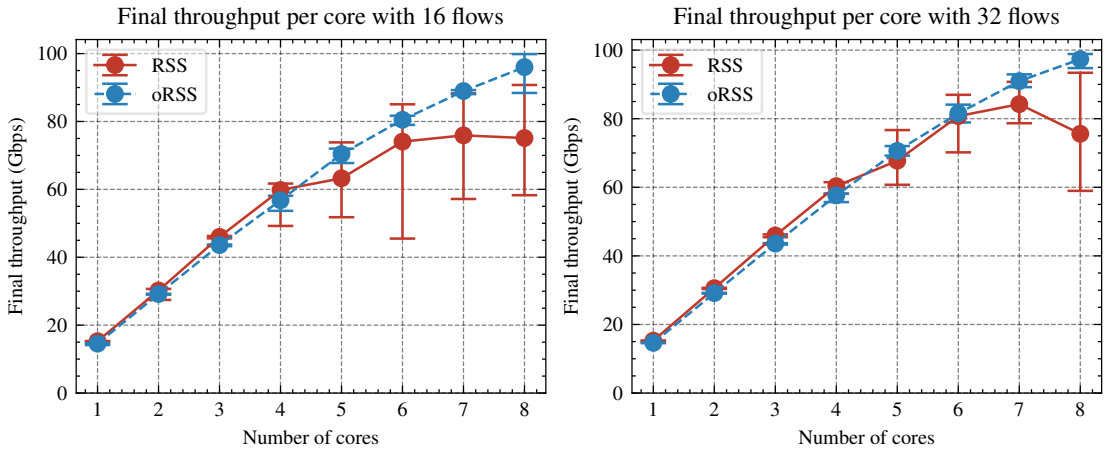


Figure 5.3: CPU profiling for oRSS-NIC with a 32 flows workload



(a) Difference between RSS and oRSS when running with 16 flows

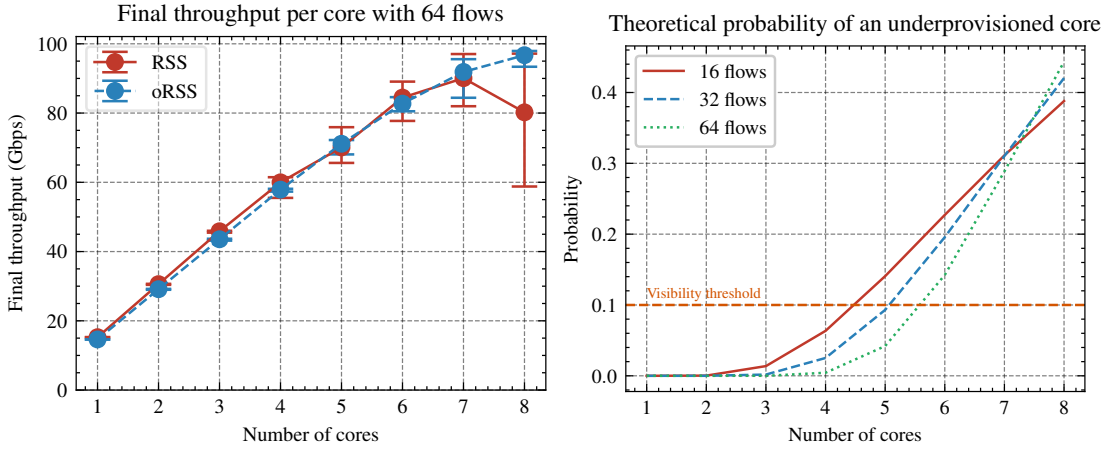
(b) Difference between RSS and oRSS when running with 32 flows

Figure 5.4: Performances comparison between RSS and oRSS

compared with figure 4.3, we can deduce that the most consuming actions would be polling, usage estimation and balancing. But the latter is expected to be rare, as it is only supposed to perform actions in situations of imbalance that it's supposed to fix, while the two others are performed at each iteration. This leads to a configuration where oRSS-NIC is most of the time simply polling and computing EMA. Furthermore, as our tests are performed with constant flows in terms of processor usage, we can expect oRSS-NIC to find optimum dispatching quickly. After this optimum is reached, almost no further migration is expected as the load is constant, further reducing the number of balancing.

5.3 Throughput

To obtain the following results, oRSS and RSS have been run over 1 to 8 cores with 16, 32 and 64 flows for a total of 48 different configurations. Each of these configurations has been tested 10 times in order to obtain representative results. Each test is running during 30 seconds. Flows have been deliberately limited to a maximum of 64 to bring forward the effects of balancing. With many flows, the law of large numbers ensures a balanced throughput by design in our current test configuration.



(a) Difference between RSS and oRSS when running with 64 flows

(b) RSS theoretical probability to have an underprovisioned core

Figure 5.5: Performances comparison between RSS and oRSS

5.3.1 RSS

As described previously, flows are of equal loads and the total workload is designed to require all 8 cores at full capacity to be completely handled. Consequently, when the host is receiving n flows while running with c cores, each core must receive $\frac{n}{8}$ flows in order to be full, and the probability for a core to receive a given flow is $\frac{1}{c}$. Thus, the probability for a core to be under-provisioned can be represented as a Binomial: $Pr(Bin(n, \frac{1}{c}) < \frac{n}{8})$, visible in figure 5.5b. Under 4 cores, the probability of getting an underloaded core is fairly below 0.1, making it unlikely in our experiment based on 10 runs. At 4 cores, this event gets close to 0.1 in the 16 flows situation, explaining the little tail observable in figure 5.4a. At 5 cores, the under-load event is expected to happen at least once in both 16 and 32 flow configurations (visible on figures 5.4b and 5.5a), generating a wider interval visible on the graph. For the remaining cores, the underload probability keeps increasing, reaching the 0.4 threshold at 8 cores, making the system under-provisioned almost half of the time.

The direct consequence of this under-provisioning is an incapacity for RSS to scale linearly in our testing configuration, as it needs all cores at high capacity in order to handle the network load. This causes a settlement between 6 and 8 cores, where the increase in computing power induced by a new core is absorbed by the loss caused by underloaded cores.

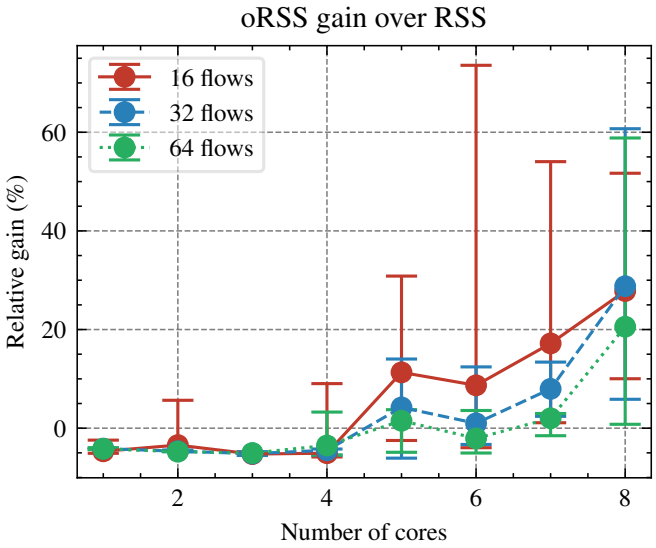
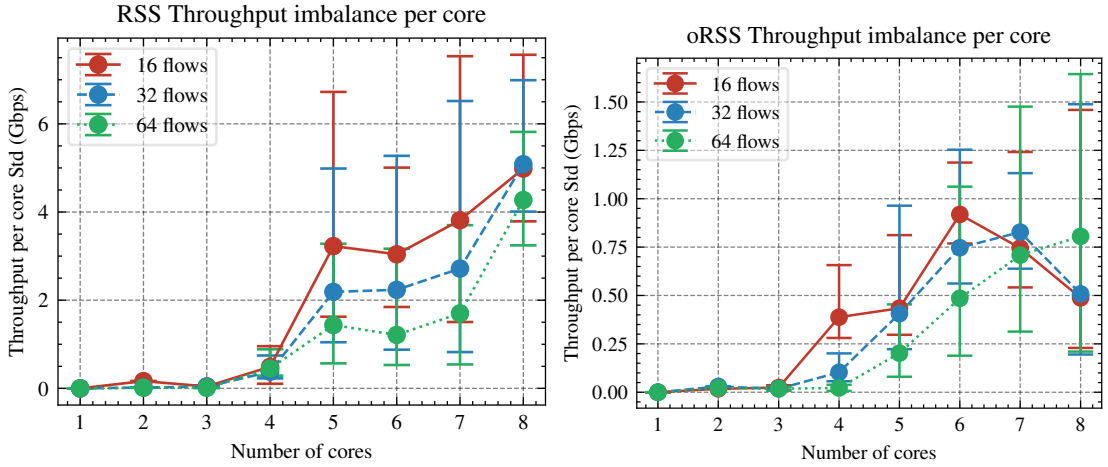


Figure 5.6: Shows the difference of throughput between RSS and oRSS. The overhead causes an initial penalty for oRSS that is quickly absorbed by a more balanced flow dispatching.

5.3.2 oRSS

On the other hand, figures 5.4a, 5.4b and 5.5a show that oRSS successfully takes advantage of its balancing mechanism in order to scale almost linearly. Globally, oRSS suffers much less from incertitude, causing smaller value intervals in each configuration from 4 cores and above compared to RSS. Nevertheless, small incertitude is still visible in configurations that are getting close to 8 cores. This can be explained by the core reduction mechanism explained in section 4.5.2. Overloaded cores would be considered impossible to balance and left unrelieved. This is expected to happen in situations with highly processor-consuming flows, as it is the case in the 16 flows configuration running on 8 cores, where each flow is consuming half a core.

Figure 5.6 compares RSS and oRSS by showing the throughput gain between the two. Under 5 cores, oRSS is penalized by its overhead and cannot match RSS which is scaling linearly. But from 6 cores to 8, this overhead is absorbed and becomes negligible compared to RSS settlement visible in all configurations. This allows oRSS to benefit from more than a 20% throughput increase compared to RSS when running on 8 cores.



(a) RSS imbalance

(b) oRSS imbalance

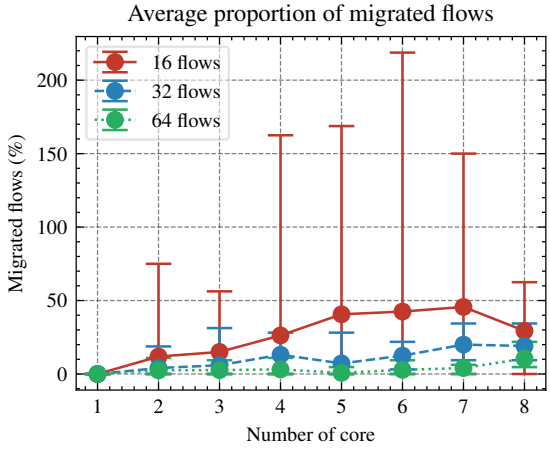
Figure 5.7: Illustration of throughput imbalance as throughput standard deviation. When close to 0, the throughput is the result of equal work from all cores. When increasing, some cores are responsible for a more important part of the throughput than others.

5.3.3 Throughput imbalance

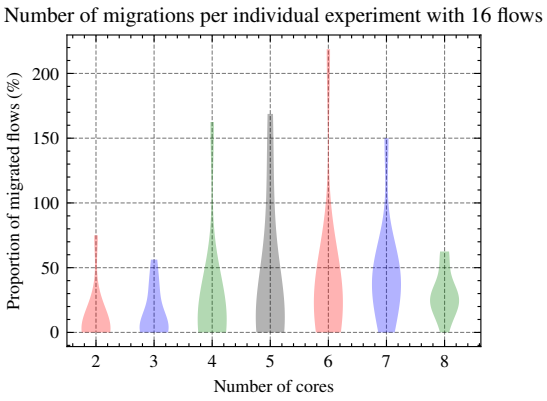
As a final note on what concerns the throughput, figure 5.7 describes the throughput imbalance between RSS and oRSS. Both exhibit an imbalance that is almost null when running with less than 3 cores, as expected by the linear scaling visible in figures 5.4a, 5.4b and 5.5a. Starting from 4 cores, an imbalance arises in both situations, but at significantly different scales. In the case of oRSS (figure 5.7b), this imbalance peaks at 1.65 Gbps in a worst-case scenario and 0.75 Gbps on average when running with 8 cores. On the other hand, RSS peaks at over 7 Gbps and 5 Gbps on average with the same configuration. With the help of its balancing algorithm, oRSS is thus capable of providing an imbalance that can be more than 6 times less important than RSS.

5.4 Migrations

Figure 5.8a allows estimating migration efficiency. An optimal migration strategy should be fairly reduced, as flows are initially randomly dispatched throughout the cores. Globally, we can observe that CPU-heavy flows tend to produce more migrations. With 16 flows, each flow consumes half a core. When more than 2 flows arrive at the same core, it causes losses. As oRSS-NIC computes CPU usage by regularly polling which packet oRSS-Host is processing, the core usage estimation



(a) Proportion of migrated flows depending on the number of running cores.



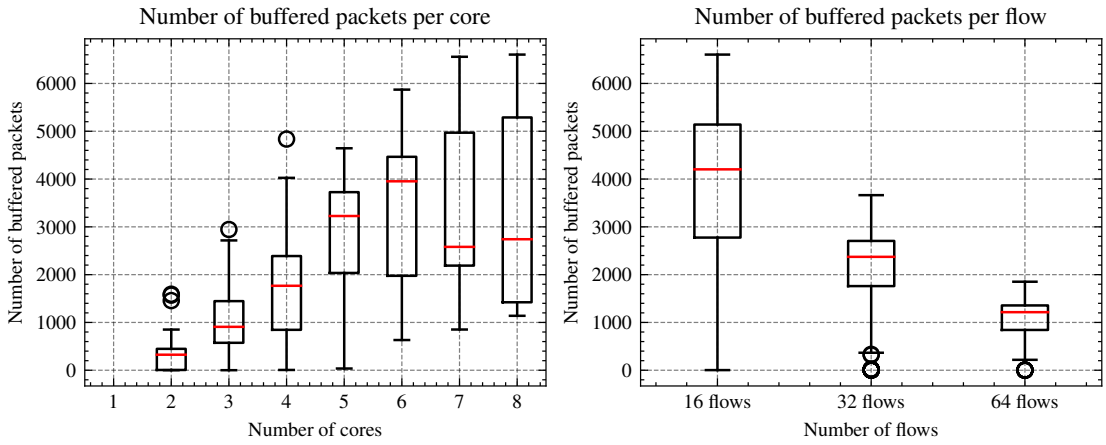
(b) Result of each test ran with 16 flows. High values are caused by a few experiments generating a lot of migrations.

Figure 5.8: Illustrations of the total number of migrations. We can see that it is relatively low, except for a few experiments.

doesn't take into account lost packets. As these losses are random, a heavy flow can appear light because most of its packets are dropped. Consequently, when such a flow is migrated, it will arrive at another less loaded core. Packets that were previously dropped are now processed and the usage of the flow increases. This can lead to 2 different situations :

- The core has enough room to process all the packets, creating imbalance but allowing a correct evaluation of the flow's weight so that it can be migrated adequately.
- The core hasn't enough room remaining, causing packet losses. This circles back to the problem, with an overloaded core providing inaccurate usage estimation.

oRSS seems to be able to loop on this problem for a while, as worst scenarios show that the number of migrations can be up to twice the actual number of flows. But this must be considered in light of the execution time of 30 seconds. With hundreds of iterations per second, exhibiting a total number of migrations of approximately 32 still witnesses that oRSS can still find an equilibrium despite initial inaccuracies. Moreover, figure 5.8b shows that such situations are extreme cases far away from most common cases.



(a) Buffered packets depending on the number of cores, all flows combined.

(b) Buffered packets per flow, all cores combined.

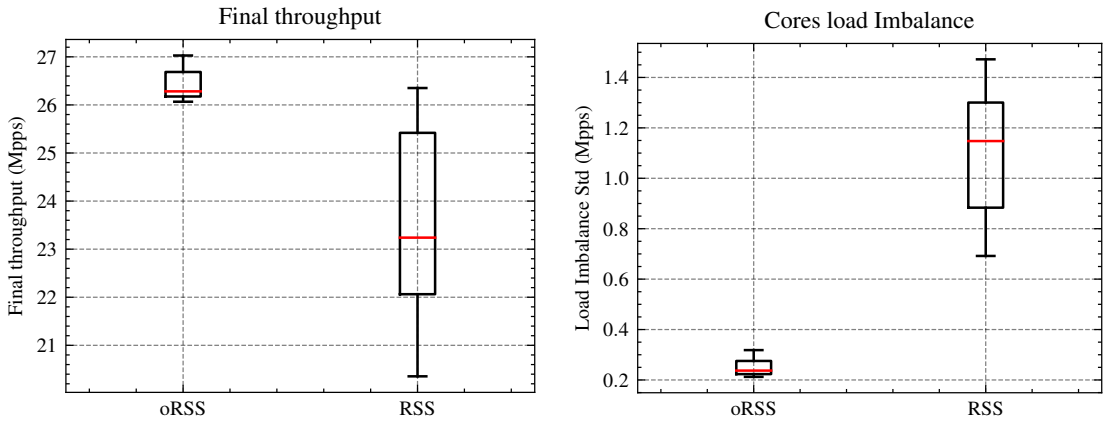
Figure 5.9: Box plots showing that the amount of buffered packets is dependent on the number of packets per flow and the number of running cores.

5.5 Buffering

Finally, we'll evaluate oRSS through the angle of packet buffering during migrations. Figure 5.9 illustrates how packet buffering happens within oRSS. We can see that it depends on two variables: the number of packets per flow and the number of running cores. The first can be easily explained, as the number of buffered packets is equal to the number of packets received during the buffering time. This is visible in figure 5.9b. The second can seem obscure if not compared with figure 5.3. We saw that oRSS-NIC spends most of its time polling and estimating host CPU usage, and both these activities are performed for each running core. As we add cores, the time required for the SmartNIC to poll and compute this usage increases, therefore reducing its iteration rate. Section 4.6.2 explained how oRSS-Host and oRSS-NIC cooperate in order to achieve a migration without packet reordering. If NIC's iteration rate drops, it'll increase the time required to perform a migration, leading to more buffered packets. This causes a linear increase in the worst-case scenario, visible in figure 5.9a.

5.6 Practical Use case

While previous sections of this chapter provide a glimpse of performance differences between oRSS and RSS, they are systematically evaluated through an artificial workload. This section aims at evaluating oRSS through a tangible application.



(a) Throughput differences between oRSS and RSS. oRSS provides on average the same performances RSS provides in its best case.

(b) Imbalance comparison between oRSS and RSS. We can see that oRSS applies a better dispatch of the workload between running cores.

Figure 5.10: Stateful Load Balancer tests results

5.6.1 Application details

The tested application is a stateful load balancer. It consists of an application trying to match patterns on the payload of each packet and forwarding them accordingly. On the first packet of a flow, the application assigns it two 8-bit values, one for matching packets and the other for unmatched ones. These values are kept in memory for each subsequent packet of the flow and are used to modify the last bit of the destination IP address depending on whether a packet is matched.

As iterating on the payload is costly in terms of resources, we ran this application over 8 cores. We fed them with 27Mpps of a mixed workload combining 50 bytes and 150 bytes packets representing respectively 80% and 20% of the workload. All these packets are dispatched through 64 flows. Other parameters are left unchanged and are identical to those described in section 5.1.

5.6.2 Results

Figure 5.10 brings results to corroborate what was already discussed in previous sections. Figure 5.10a shows how a better flow dispatching among cores can result in increased throughput and reduced incertitude as the oRSS value range is much smaller than RSS's. Figure 5.10b illustrates such dispatching, proving that oRSS's rebalancing mechanism reduces the load imbalance between running cores.

Chapter 6

Conclusion

This document presented oRSS, an offloaded alternative to RSS that leverages kernel bypass and SmartNICs in order to cope with important network bandwidth. Through a careful migration system, oRSS is capable of balancing flows among end-host cores to maximize available performances.

We dived into oRSS's inner workings to describe how it gathers information about currently processed flows, computes an estimation of per-flow CPU usage, decides if a balancing must be performed and how it actually applies this balancing without causing packet reordering.

Through evaluation, we showed that oRSS scales almost linearly to reach the 100Gbps linerate while RSS stumbles on unbalanced flows dispatching, keeping it around 80Gbps when running on 8 cores. Thanks to offloading, the light overhead of oRSS is quickly absorbed by a better flow dispatch, allowing it to outperform RSS in our test configuration. Furthermore, we investigated migrations to see that oRSS eventually finds an equilibrium, even if it doesn't always do so optimally. Finally, this document analyzed packet buffering, showing that it depends on the number of packets per flow and the number of running cores.

This work shows that it is possible to take advantage of SmartNICs to offload balancing on less performant ARM, providing both a better work balance between cores and freeing precious CPU cycles that can be dedicated to actual data processing.

Chapter 7

Future Work

There is a number of directions in which we could improve oRSS.

Comparing oRSS and RSS++
While this document addresses how oRSS can outperform RSS, comparing oRSS and RSS++ could provide an interesting benchmark to separate the performance gains of offloading and balancing. As oRSS relies on a more complex and costly architecture, being able to quantify performance gains in moving from RSS++ to oRSS could be useful.

Software optimizations
At its current stage, oRSS is thought of as a proof-of-concept. An interesting research could be to investigate how much software optimizations such as multicore processing, asynchronous DMA requests, etc. can improve its performances.

Connection Management
In the current situation, oRSS doesn't handle connection teardowns, working with infinite flows. This would be an important feature to allow oRSS to work in a more realistic environment.

Avoiding second VLAN parsing overhead
As described in Chapter 5, oRSS currently suffers from a hardware limitation keeping it from parsing two VLANs in hardware. This forces us to parse the connection id manually, causing a certain overhead. An improvement could be to either test oRSS on another SmartNIC or to find another method to signal the connection id to the DPDK application.

Bibliography

[1] Tom Barbette et al. "RSS++: Load and State-Aware Receive Side Scaling". In: *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. CoNEXT '19. Orlando, Florida: Association for Computing Machinery, 2019, 318–333. ISBN: 9781450369985. DOI: 10.1145/3359989.3365412. URL: https://doi.org/10.1145/3359989.3365412.

[2] Adam Belay et al. "IX: a protected dataplane operating system for high throughput and low latency". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 49–65.

[3] Qizhe Cai et al. "Understanding Host Network Stack Overheads". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. Virtual Event, USA: Association for Computing Machinery, 2021, 65–77. ISBN: 9781450383837. DOI: 10.1145/3452296.3472888. URL: https://doi.org/10.1145/3452296.3472888.

[4] Ruining Chen and Guoao Sun. "A Survey of Kernel-Bypass Techniques in Network Stack". In: *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*. CSAI '18. Shenzhen, China: Association for Computing Machinery, 2018, 474–477. ISBN: 9781450366069. DOI: 10.1145/3297156.3297242. URL: https://doi.org/10.1145/3297156.3297242.

[5] Tianyi Cui et al. "Offloading Load Balancers onto SmartNICs". In: *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '21. Hong Kong, China: Association for Computing Machinery, 2021, 56–62. ISBN: 9781450386982. DOI: 10.1145/3476886.3477505. URL: https://doi.org/10.1145/3476886.3477505.

[6] DPDK. *DPDK MLX5 Compatibility*. https://doc.dpdk.org/guides/nics/mlx5.html.

[7] Faucet. *Ryu SDN Framework*. https://ryu-sdn.org/. 2023.

[8] Liang Guo and I. Matta. “The war between mice and elephants”. In: *Proceedings Ninth International Conference on Network Protocols. ICNP 2001.* 2001, pp. 180–188. DOI: 10.1109/ICNP.2001.992898.

[9] Jack Tigar Humphries et al. “Mind the Gap: A Case for Informed Request Scheduling at the NIC”. In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. HotNets ’19. Princeton, NJ, USA: Association for Computing Machinery, 2019, 60–68. ISBN: 9781450370202. DOI: 10.1145/3365609.3365856. URL: https://doi.org/10.1145/3365609.3365856.

[10] Intel. “Improving Network Performance in Multi-Core Systems”. In: White paper, 2016.

[11] Anuj Kalia, Michael Kaminsky, and David G. Andersen. “Design Guidelines for High Performance RDMA Systems”. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 437–450. ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[12] Magnus Karlsson and Björn Töpel. “The path to DPDK speeds for AF XDP”. In: *Linux Plumbers Conference*. 2018.

[13] Jakub Kicinski and Nicolaas Viljoen. “eBPF Hardware Offload to SmartNICs: cls bpf and XDP”. In: *Proceedings of netdev* 1 (2016).

[14] Linux Foundation. *Data Plane Development Kit*. https://www.dpdk.org/. 2023.

[15] Jianshen Liu et al. “Performance Characteristics of the BlueField-2 SmartNIC”. In: *CoRR* abs/2105.06619 (2021). arXiv: 2105.06619. URL: https://arxiv.org/abs/2105.06619.

[16] Nick McKeown et al. “OpenFlow: Enabling Innovation in Campus Networks”. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (2008), 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: https://doi.org/10.1145/1355734.1355746.

[17] Nvidia. *ASAP²*. https://network.nvidia.com/files/doc-2020/sb-asap2.pdf. 2019.

[18] NVIDIA. *OVS Offload Using ASAP2 Direct*. https://docs.nvidia.com/networking/display/MLNXENv531001/OVS+Offload+Using+ASAP2+Direct.

[19] Amy Ousterhout et al. “Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads.” In: *NSDI*. Vol. 19. 2019, pp. 361–378.

[20] Ben Pfaff et al. “The Design and Implementation of Open vSwitch”. In: *12th USENIX symposium on networked systems design and implementation (NSDI 15)*. 2015, pp. 117–130.

[21] George Prekas, Marios Kogias, and Edouard Bugnion. “Zygos: Achieving low tail latency for microsecond-scale networked tasks”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 325–341.

[22] Alexander Rucker et al. “Elastic rss: Co-scheduling packets and cores using programmable nics”. In: *Proceedings of the 3rd Asia-Pacific workshop on networking 2019*. 2019, pp. 71–77.

[23] Hugo Sadok et al. “We need kernel interposition over the network dataplane”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2021, pp. 152–158.

[24] Anirudh Sivaraman et al. “Towards programmable packet scheduling”. In: *Proceedings of the 14th ACM workshop on hot topics in networks*. 2015, pp. 1–7.

[25] Delzotti, Clément. *offloaded Receive Side Scaling (oRSS): source code*. `https://forge.uclouvain.be/orss`. 2023.

[26] IEEE. *IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks*. `https://standards.ieee.org/ieee/802.1Q/1039/`. 1998.

[27] Open Networking Foundation. *OpenFlow Switch Specification, version 1.3*. `https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf`. 2012.

[28] Shaoke Xi, Fuliang Li, and Xingwei Wang. “FlowValve: Packet Scheduling Offloaded on NP-based SmartNICs”. In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2022, pp. 347–358.

[29] Irene Zhang et al. “I’m Not Dead Yet! The Role of the Operating System in a Kernel-Bypass Era”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS ’19. Bertinoro, Italy: Association for Computing Machinery, 2019, 73–80. ISBN: 9781450367271. DOI: 10.1145/3317550.3321422. URL: https://doi.org/10.1145/3317550.3321422.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**