

On the Performance of KVM-Based Virtual Routers

Luca Abeni^{a,*}, Csaba Kiraly^b, Nanfang Li^c, Andrea Bianco^c

^a*DISI – University of Trento, Via Sommarive 5, Trento (TN), Italy*

^b*Bruno Kessler Foundation, Via Sommarive 18, Trento (TN), Italy*

^c*Dipartimento di Elettronica e delle Telecomunicazioni, Politecnico di Torino, Torino, Italy*

Abstract

This paper presents an evaluation of the performance of software virtual routers, by analysing the packet forwarding capabilities of the Linux kernel running inside a KVM virtual machine and the performance of more advanced architectures based on virtual routers aggregation. More specifically, the performance impact of various tuning and configuration options is evaluated with focus on the virtual router interconnection mechanism, the algorithm used for scheduling the virtual router tasks, the number of CPU cores and their CPU bindings. The presented results show how to properly configure the virtual router components to improve the virtual routing performance and how to control the performance by using proper CPU schedulers for desired quality of service. Moreover, some modular architectures which can better exploit the available CPU cores are also presented, showing how to reach performance similar to the ones of non-virtualised software routers.

1. Introduction

Software Routers (SRs), i.e. routers implemented by software running on commodity off-the-shelf hardware, became in recent years an appealing solution compared to traditional routing devices based on custom hardware. SRs' main advantages include cost (the multi-vendor hardware used by SRs can be cheap, while custom equipments are more expensive and imply higher training investment), openness (SRs can be based on open-source software, and thus make use of a large number of existing applications) and flexibility. Since the forwarding performance provided by SRs has historically been an obstacle to their deployment in production networks, recent research works focused on increasing SRs performance by either using massively parallel hardware such as a GPU to process packets [1], allowing the routing software to directly access the networking hardware (thus eliminating the overhead introduced by the OS kernel) [2], or using other similar techniques to improve the forwarding performance of monolithic routers. An orthogonal approach to improve SR performance can be based on the aggregation of multiple devices to form a more powerful

*Corresponding Author

Email addresses: luca.abeni@unitn.it (Luca Abeni), kiraly@fbk.eu (Csaba Kiraly), nanfang.li@polito.it (Nanfang Li), andrea.bianco@polito.it (Andrea Bianco)

routing unit like the Multistage Software Router [3], Router Bricks [4] and DROP [5]. While by improving the performance of a single routing device it is possible to reach the forwarding speed of multiple tens of Gigabit per second [1], the aggregation of multiple routing units can allow the forwarding speed to scale almost linearly with the number of used devices [3].

As recognised by several researchers [6, 7], virtualisation techniques could become an asset in networking technologies, improving SRs flexibility and simplifying their management. As an interesting example, the live migration capability provided by some Virtual Machines (VMs) could be adopted for consolidation purposes and/or to save energy. Moreover, running a SR in a VM allows to dynamically adapt the forwarding performance of the (virtual) device to the workload by renting virtual resources instead of buying new hardware. This feature is especially useful when the network traffic has a high variance, thus a high processing power might be necessary only for short periods. Virtualisation can also simplify the management of a SR, and improve its reliability: for example, migration of VMs during maintenance periods can be implemented and faster reaction to failures should be expected by booting new VMs on general purpose servers. Finally, the same physical infrastructure can be sliced and shared among different users to improve hardware efficiency.

Obviously, the usage of Virtual Software Routers (VSRs) might increase the complexity of the routing software: for example, the communications between VMs and the physical nodes hosting them result in complex interactions between hardware and VMs, which could easily compromise VSR's performance. This paper focuses on analysing such interactions to identify and remove various performance bottlenecks in the implementation of a VSR. Since such an investigation is easier when the behaviour of and the interaction mechanisms among all the software components (routing software, virtual machine software, operating system kernel, etc...) are known, this work focuses on an open-source virtualisation environment (KVM, the Kernel-based Virtual Machine [8]) which permits to easily analyse the VM and SR behaviour to identify performance bottlenecks. Indeed, since KVM is tightly integrated into Linux, it is possible to exploit all the available Linux management tools. Furthermore, KVM has been included into the Linux mainline from 2.6.20 on.

VSR performance can be improved by carefully tuning various system parameters such as those related to the connection mechanisms among VMs, threads priorities and CPU affinities. Some preliminary results [9] show that a proper configuration and optimisation of the virtual routing architecture and the aggregation of multiple VSRs (as suggested by the multistage software router architecture [3]) permit to forward about 1200kpps (with 64 bytes packets) in a commodity PC, close to the physical speed of a Gigabit Ethernet. This paper extends the preliminary results investigating the bottlenecks for reaching the full line rate, by comparing different technologies for interfacing the VMs with the physical hardware, and by showing how a proper CPU scheduler can allow to control the performance of a VSR (and to limit its CPU usage).

2. Virtualisation Technologies

VSR performance is obviously affected by the amount of computational resources i) available on the physical node hosting the VSR and ii) needed by the VSR to route packets. Computational resources are mainly used by:

1. the forwarding code in the SR (named *guest* because it is the code running *inside* a VM). This code can be the packet forwarding subsystem of the Linux kernel, or Click, or some other kind of SR code;
2. the physical machine hosting the VM (called *host*) to move packets among physical interfaces, virtual switches, and virtual interfaces of guests.

Host to/from VM packet forwarding (item 2) can be executed in the OS kernel, in an hypervisor, or in some user-space component (typically, the Virtual Machine Monitor - VMM), depending on the specific virtualisation architecture.

If the VSR is implemented using a “closed” virtualisation architecture such as VMWare [10], it is not easy to understand how much CPU time is consumed by the VMM, by the guest, or by the host OS kernel. Hence, in this paper an open-source virtualisation architecture is used. The two obvious candidates are Xen [11] and KVM [8]. Since the KVM architecture is more similar to the standard Linux architecture (hence, it does not require to learn new profiling and performance evaluation tools), it has been selected for running the experiments presented in this paper. KVM is based on a kernel module, which exploits the virtualisation features provided by modern CPUs to directly execute guest code, and on a user-space VMM, based on QEMU [12], which virtualises the hardware devices and implements some virtual networking features.

The most relevant feature for VSRs provided by the user-space VMM is the emulation of network interfaces, because CPU virtualisation is not an issue, as KVM allows guest machine instructions to run at almost-native speed. When a packet is received, the VMM reads it from a device file (typically the endpoint of a TAP device) and inserts it in the ring buffer of the emulated network card (the opposite happens when sending packets). When emulating a standard network interface (such as an Intel e1000 card), the VMM moves packets to/from the guest by emulating all hardware details of a real network card. This process is time consuming, easily causing poor forwarding performance, especially when considering small packets, and/or high interrupt rates. This issue can be addressed by using `virtio-net`¹, which does not emulate real hardware but uses a special software interface to communicate with the guest (that needs special `virtio-net` drivers). Thus, the overhead introduced by emulating networking hardware is reduced, and forwarding performance is improved. The para-virtualised NIC is based on a ring of buffers shared between guest and VMM, which can be used for sending/receiving packets. Guest and VMM notify each

¹`virtio-net` is a para-virtualised I/O framework for high speed guest networking [13]

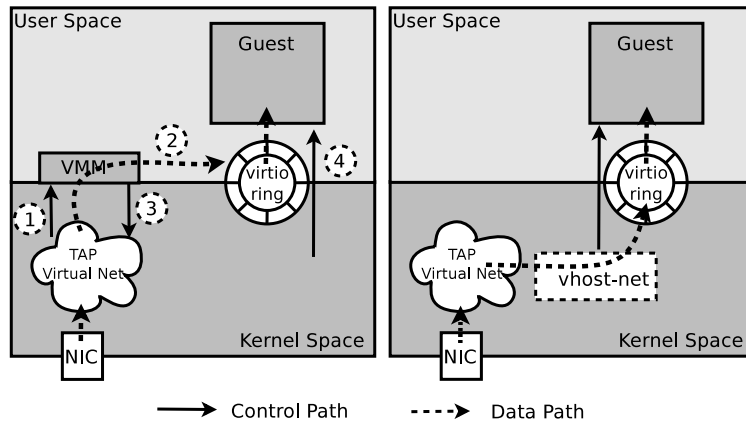


Figure 1: The VSR packet path comparison between `virtio-net` (left) and `vhost-net` (right).

other when buffers are empty/full, and the `virtio-net` mechanism is designed to minimise the amount of host/guest interactions (by clustering the notifications, and allowing to transfer data in batches).

When using `virtio-net`, the user-space VMM is still responsible for moving data between the (endpoint of the) TAP interfaces and the `virtio-net` ring buffers. Hence, when a packet is received, as depicted in the left side of Fig. 1:

1. the host kernel notifies the user-space VMM that a new packet is available on the TAP device file;
2. the VMM is scheduled, reads the available packets from the device file and copies them to the `virtio-net` ring;
3. the VMM notifies the guest and execution returns to the kernel;
4. the guest receives a `virtio-net` interrupt, starts executing and can receive and process the packet.

In summary, the large number of switches between host kernel, VMM, and guest, can introduce overhead and decrease virtual router performance. This problem can be solved by using `vhost-net`², which is a helper mechanism provided by the host kernel, able to directly copy packets between the TAP interface and the `virtio-net` ring buffers. Thus, the copy is not performed by the user-space VMM, but by a dedicated kernel thread (referred as “the `vhost-net` kernel thread” from now) and some context switches can be avoided. As a result, forwarding performance of the guest are largely improved.

When using `vhost-net`, the user-space VMM does not need to execute when the guest sends and receives network packets, and the CPU time consumed by the host to move packets is not used by the user-space VMM but by the `vhost-net` kernel thread (notice that there is one `vhost-net` kernel thread per virtual interface), as shown in the right hand side of Fig. 1. The guest code executes in a different thread, named `vcpu` thread (notice that there is one `vcpu` thread per virtual CPU).

²<http://www.linux-kvm.org/page/VhostNet>

More details about how packets are moved between physical and virtual interfaces are available in the Appendix.

3. Optimising a Monolithic Virtual Router

To understand how the various mechanisms described in the previous sections affect the performance of a VSR, and how to correctly configure the host and the VMM to optimise the virtual routing performance, a set of experiments were performed on the simplest possible VSR implementation first. The VSR used for these experiments is composed by a Linux-based OS running inside a KVM-based VM, and is referred to as *monolithic* VSR in the rest of this paper. Thanks to the fact that KVM is an open virtualisation architecture, it is possible to identify the performance bottlenecks of a VSR and to understand how to exploit existing resources to tune and improve the virtual routing performance.

To simplify the setup, these first experiments are based on a testbed composed by a traffic generator and one VSR node. To verify that the results are not strictly dependent on a specific hardware, the experiments were repeated using different hardware. The results obtained using different hardware (different kinds of x86 CPUs, different amounts of RAM, and different kinds of Intel-based Gigabit Ethernet cards) are consistent: although there is some performance difference at the peak rate, the qualitative behaviour is the same in all the different setups. Hence, for the sake of simplicity, the paper reports the results obtained using an Intel Xeon quad core E5-1620 running at $3.66GHz$.

3.1. Exploiting CPU Cores

In the first experiment, the performance of a monolithic VSR is measured when forwarding small packets (64 bytes long). Fig. 2 shows the forwarding packet rate (as a function of the input packet rate) of a 3.4 Linux kernel running inside `qemu-kvm` 1.1.0. Each experiment has been repeated 10 times, and the 99% confidence intervals are also reported. Since this CPU has multiple cores, the VSR was tested while restricted to use a single core (all interrupts are processed on the first CPU core, where the KVM `vcpu` thread and the `vhost-net` kernel thread also run), when restricted to 2 cores (interrupt processing and threads execution on the first 2 CPU cores), and when all 4 CPU cores are available (only 3 CPU cores are used: one for interrupt processing, one for the KVM `vcpu` thread, and one for the `vhost-net` kernel thread).

Using multiple cores improves VSR performance, as shown in Fig. 2. However, the performance improvement can be achieved only if the `vcpu` thread and the `vhost-net` kernel thread are properly scheduled. For example, increasing the number of used CPU cores from 2 to 4 resulted in no significant performance improvement if the two threads were left free to execute on any CPU core (“4 cores, no bind” line). This is probably due to a deficiency in the load balancer used by the Linux CPU scheduler, which seems to ignore the CPU loads caused by interrupt processing. This issue was addressed by setting the affinity of the two

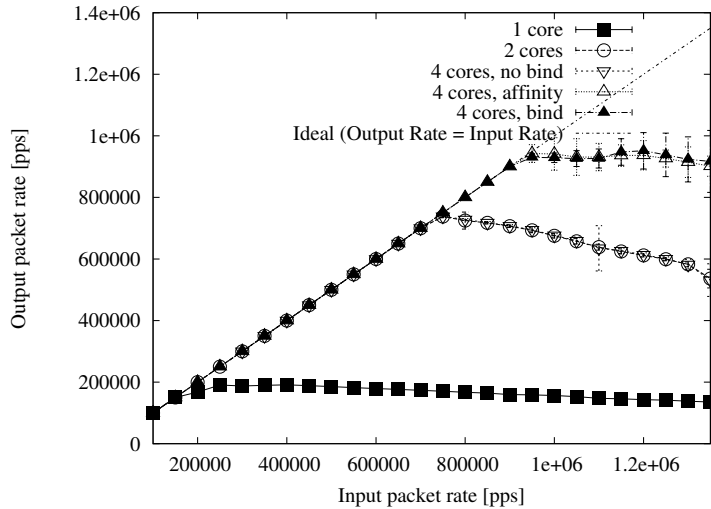


Figure 2: Forwarding performance of a monolithic VSR while increasing the number of CPU cores.

threads to cores 1, 2, and 3, so that they are not executed on core 0, which is busy with interrupt processing. The results achieved by using this assignment are shown in the “4 cores, affinity” line. More strict CPU bindings for the `vcpu` and `vhost-net` threads (allowing the `vcpu` thread to execute only on core 1, and the `vhost-net` kernel thread to execute only on core 2) allowed to reduce the variance (and the confidence interval), but did not have any particular impact on the average performance (see the “4 cores, bind” line). Notice that even when 4 CPU cores are made available (and `top` shows a high amount of idle CPU time in the system), the virtual router is not able to forward more than $900kpps$.

By analysing the system in overload (for input rates larger than $900kpps$), it becomes clear that the bottleneck is the `vhost-net` kernel thread, which consumes all the CPU time on a core (see Sec. 3.3). Since the issue is that a single thread (the `vhost-net` thread) needs more than 100% of the CPU time of a single core, playing with CPU bindings cannot help anymore, because a single thread cannot simultaneously execute on 2 different CPU cores. Thus, it is not possible to exploit the huge amount of idle time on other cores.

3.2. Setting the Thread Priorities

When some of the threads implementing the monolithic VSR are executed on the same CPU core, their scheduling priorities can have a huge impact on virtual routing performance. Fig. 3 shows the results achieved with a monolithic VR when the 2 important threads implementing the VR (the KVM `vcpu` thread and the `vhost-net` kernel thread) are bound to the same CPU core. The two threads are scheduled through fixed priorities by using the `SCHED_FIFO` scheduling class, changing their priority order between experiments. Notice that interrupt handlers are executed on a different CPU core, and this is a key difference with respects

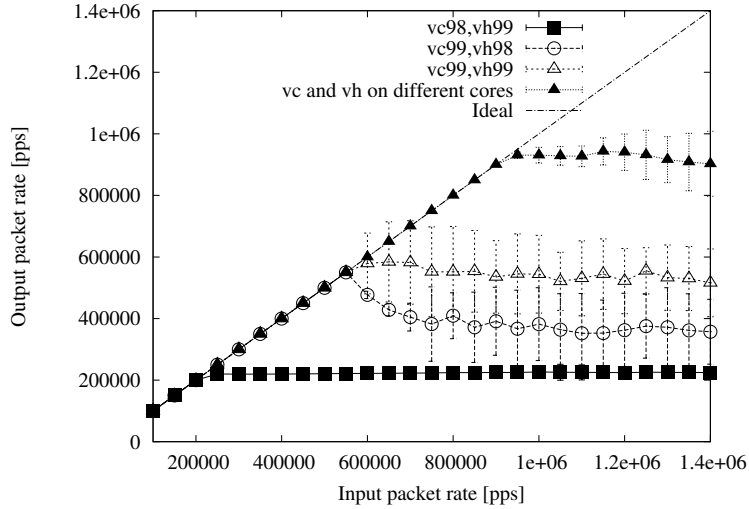


Figure 3: Forwarding performance of a monolithic VSR with different priorities for the `vcpu` and `vhost-net` threads (`vcN` means `vcpu` with priority `N`, `vhN` means `vhost-net` with priority `N`).

to the “1 core” line of Fig. 2 (hence the different behaviour in overload). For reference, the performance curve obtained scheduling `vcpu` and `vhost-net` on 2 separate cores (equivalent to the “4 cores, bind” curve in Fig. 2) is also shown in the figure.

The worst forwarding performance is achieved when the `vhost-net` kernel thread has a higher priority than the `vcpu` thread (“`vc98,vh99`” line). This happens because the `vhost-net` is responsible for both moving the received packets from the physical NIC to the VM and moving back the routed packets from the VM to the physical NIC. If this thread has a higher priority than the `vcpu` thread, most of the core’s time is spent moving to the VM input packets that the `vcpu` thread is not able to route back (because it is starved by `vhost-net`). On the contrary, a higher priority for the `vcpu` thread can guarantee that each packet moved from the physical NIC to the VM by `vhost-net` can be correctly routed back, hence the peak throughput increases (see the “`vc99,vh98`” line). However, when the system is overloaded the `vcpu` thread risks to starve the `vhost-net` kernel thread, so an increasing number of the routed packets cannot be moved back to the physical NIC. This is why the performance decrease in overload. As a result, the best forwarding throughput is obtained when assigning the same priority to the two threads (“`vc99,vh99`” line).

3.3. Multiple `vhost` and `vcpu` Threads Performance

By looking at how the various threads implementing a VSR use the CPU cores, it is possible to understand what happens when the peak forwarding rate is reached, and how to improve the virtual routing performance in this case.

Fig. 4 shows the CPU utilisation caused by a monolithic VSR when the network interrupts are handled on core 0, the `vcpu` thread executes on core 1, and the `vhost-net` kernel thread executes on core 2 (line “4

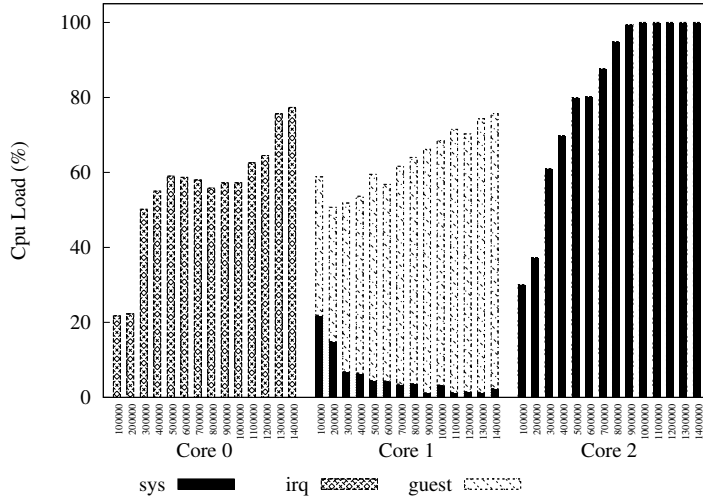


Figure 4: CPU utilisation detailed per core and per type of load, as a function of input packet rate [pps], for a monolithic VSR with 1 `vhost-net` kernel thread (bound to core 2) and 1 `vcpu` thread (bound to core 1).

cores, bind” in Fig. 2). Results are detailed per CPU core, showing how utilisation changes as a function of input packet rate (shown on the horizontal axis in pps). `sys` represents the percentage of time consumed by threads executing in kernel space (for example, the `vhost-net` kernel thread, or a user thread executing a system call), `irq` represents the percentage of time consumed by interrupt handlers (in this case, the handlers of the network interrupts), and `guest` represents the percentage of time consumed by KVM executing guest code, i.e. the `vcpu` thread. The peak rate is reached when the `vhost-net` kernel thread consumes almost all of the execution time on core 2. Hence, the performance limit is due to a `vhost-net` overload. As a consequence, it can be conjectured that the performance can be increased by using 2 `vhost-net` kernel threads and executing them on 2 cores (core 2 and core 3). This has been verified by running some additional tests with different numbers of threads bound to various CPU cores.

Fig. 5 shows the monolithic VSR performance when using a different number of threads (and, as a consequence, a different number of CPU cores), and Figs. 6 and 7 show the CPU utilisations in these experiments. Notice that the “1 `vhost` 1 `vcpu`” line is the same as the “4 cores, bind” line of Fig. 2.

These figures show that splitting the `vhost-net` kernel thread into two threads (one for packets reception, the other one for transmission) permits to exploit additional CPU cores, increasing the peak throughput to more than $900kpps$. By looking at Fig. 6, it becomes clear that when two `vhost-net` kernel threads are used, the virtual routing performance is limited by the `vcpu` thread. Interrupt handling is also near the limits, almost saturating Core 0.

Furthermore, increasing the number of `vcpu` threads without creating additional `vhost-net` kernel threads does not improve the virtual routing performance, because the performance bottleneck remains

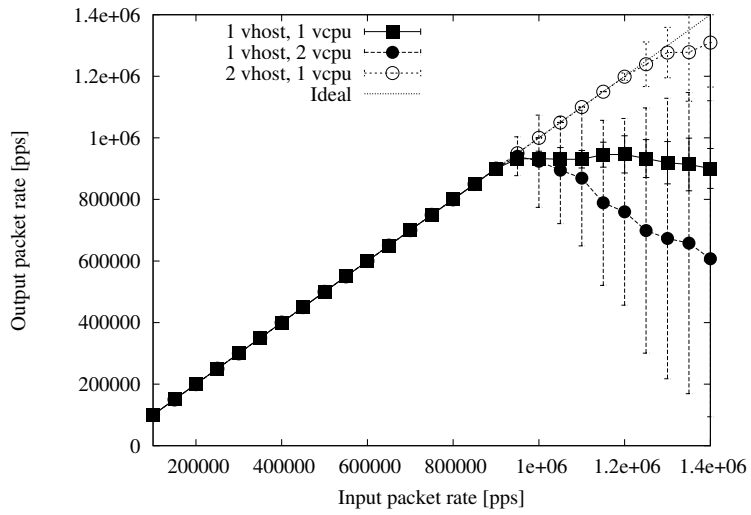


Figure 5: Forwarding performance for a monolithic VSR with different vhost and vcpu thread(s).

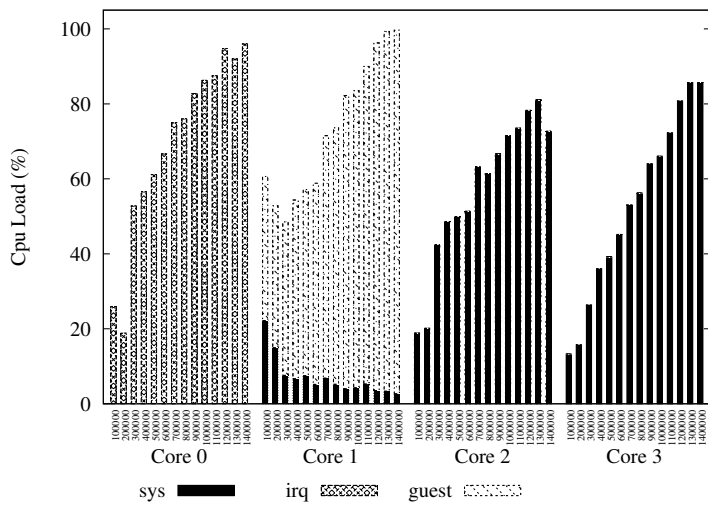


Figure 6: CPU statistics for the standard VSR, with 2 vhost and 1 vcpu threads

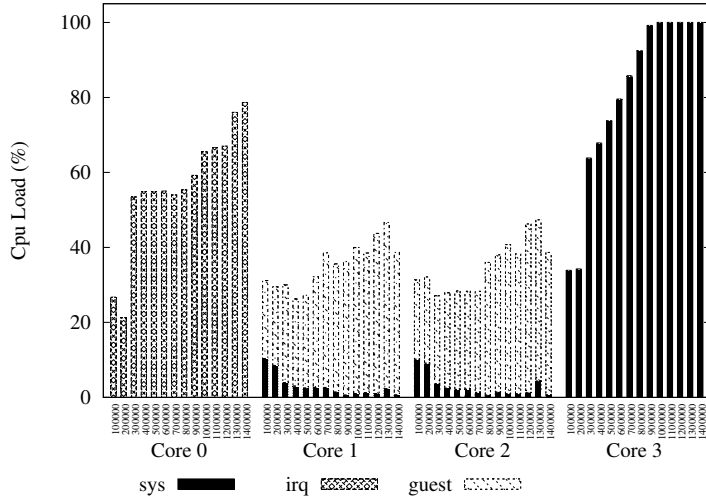


Figure 7: CPU statistics for the standard VSR, with 1 vhost and 2 vcpu threads

in the single `vhost-net` kernel thread, as shown in Fig. 7. The additional `vcpu` thread even decreases the performance in overload, probably because of the overhead (a more complex SMP VR architecture and more context switches) introduced by the additional thread.

3.4. Interconnection Mechanisms

As shown in the previous experiments, the limitations to the maximum packet forwarding rate by a VSR are often due to an overload of the `vhost-net` kernel thread, which is responsible for moving packets between the physical interface and the virtual one (and vice versa). Hence, it might be interesting to understand how the different mechanisms that can be used to connect the physical and virtual interfaces affect virtual routing performance.³ This investigation has been performed by running a new set of experiments, comparing a Linux *software bridge* (`br` device) plus TAP interface, `openvswitch` (plus a TAP interface), the Linux `macvtap` interface and `netmap` [2], a novel mechanism providing high performance in user-space networking.

The results of this comparison are shown in Fig. 8, and show that `macvtap` tends to perform better than the other mechanisms, followed by `openvswitch`, `netmap` and the software bridge. The better performance provided by `macvtap` are probably due to the fact that it combines into a single operation some of the data movements performed in the bridge and `openvswitch` configurations. Indeed, when using two separate bridge (either the “standard” software bridge or `openvswitch`) and TAP devices, packets are moved from the physical NIC to the bridge device and then from such a device to the TAP interface. `Openvswitch` performs better than the standard software bridge because it has been optimised to reduce the overhead introduced by the Linux bridge device (see also [14]).

³In the previous experiments, the `macvtap` mechanism provided by the Linux kernel has been used.

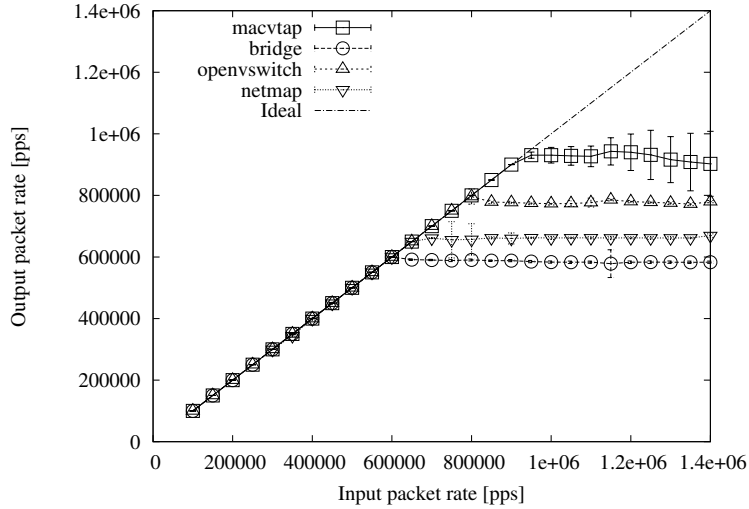


Figure 8: Forwarding performance comparison in KVM using different interconnection mechanisms.

Summing up, `macvtap` is less flexible than the other solutions but performs better and should be preferred when building a high performance monolithic VR.

While the performance difference between `macvtap` and the bridge-based solutions (standard bridge and `openvswitch`) were expected, the `netmap` results seem to contradict the original `netmap` papers (which show that `netmap` can achieve better performance than the standard mechanisms implemented in Linux). Hence a more accurate investigation has been performed to understand the reasons for this unexpected behaviour.

Notice that `netmap` has a maximum throughput around $650kpps$, the virtual routing performance do not degrade when the input packet rate increases further, and confidence interval is quite small even in overload. On the other hand, `macvtap` has a peak forwarding rate of about $900kpps$, performance slightly decrease as the input packet rate further increases, and the confidence interval in overload is larger. Hence, `netmap` seems to have a lower peak rate with respect to the `macvtap` configuration, but has a better behaviour in overload. Analysing the CPU utilisation with `top`, the source of this strange behaviour become clear. The key point is that the `macvtap` configuration used `vhost-net` for moving packets from the `macvtap` interface to the `virtio-net` device inside the VM. Since `vhost-net` creates an additional kernel thread, the `macvtap` configuration is able to use 3 CPU cores (see Fig. 4), while `netmap` never uses more than 2 CPU cores. Indeed, `vhost-net` can distribute at least 3 activities (interrupt handling, the `vhost-net` kernel thread, and the `vcpu` thread) on the CPU cores, while `netmap` only uses 2 active entities: the main `qemu` thread, which implements interrupt handling and moves the network packets in the `virtio` ring, and the `vcpu` thread (which works as in the `vhost-net` setup). Hence, `netmap` cannot take advantage of additional CPU cores. To verify these findings, the experiment was repeated comparing `netmap` with a `macvtap` configuration in which only 2 CPU cores are used. The results are reported in Fig. 9, which compares the performance

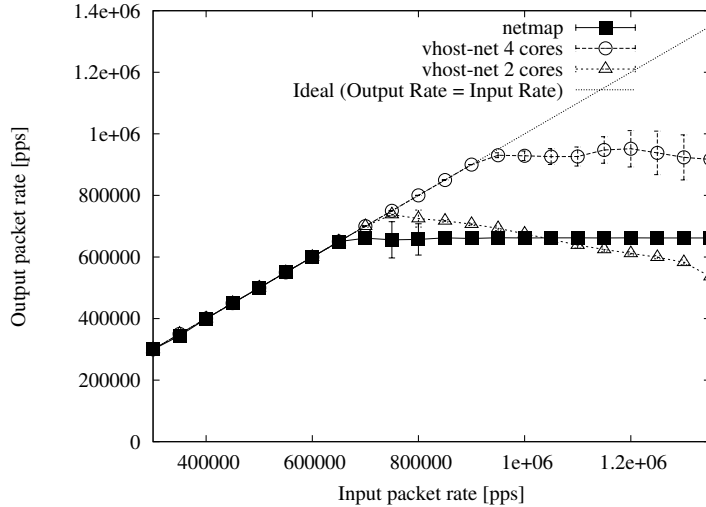


Figure 9: Forwarding performance comparison of a monolithic VSR using `vhost-net` and `netmap` configuration.

of `macvtap` (using `vhost-net`) with 2 and 4 CPU cores with the performance achieved when using `netmap` (which cannot use more than 2 CPU cores)⁴. Notice that the “`vhost-net, 4 cores`” line in Fig. 9 is equivalent to the “4 cores, bind” line of Fig. 2, and the “`vhost-net 2 cores`” in Fig. 9 is equivalent to the “2 cores” curve from Fig. 2.

If `netmap` is compared with `macvtap` plus `vhost-net` using only 2 CPU cores (“`vhost-net 2 cores`” in Fig. 9), it shows similar routing performance with a better behaviour in overload.

3.5. Controlling VSR Performance

All the results reported until now show that there is a clear relationship between the amount of CPU time consumed by a VSR (in particular, by the `vcpu` thread and by the `vhost-net` kernel thread) and VSR performance (in terms of forwarding packet rate). As a result, if the CPU scheduler is able to control the amount of CPU time consumed by a thread, then this mechanism can be used to control VSR performance instead of using the kernel’s Traffic Control or network scheduling modules, that can introduce unneeded overhead.

This has been verified by using the `SCHED_DEADLINE` scheduler for Linux [15], which implements the Constant Bandwidth Server (CBS) scheduling algorithm [16]. This scheduling algorithm permits to reserve to a specific task, thread, or process an amount of time equal to Q^s every period T^s . As a result, the task is guaranteed to be able to execute for Q^s time units every T^s , but cannot consume more than this amount of time. Thus, Q^s/T^s represents the fraction of CPU’s processing time that the task can use.

⁴Notice that `macvtap` provides the low level (i.e, NIC to TAP) connection and `vhost-net` provides the high level (i.e, TAP to guest) connection, while `netmap` can pass packets directly to user space guest.

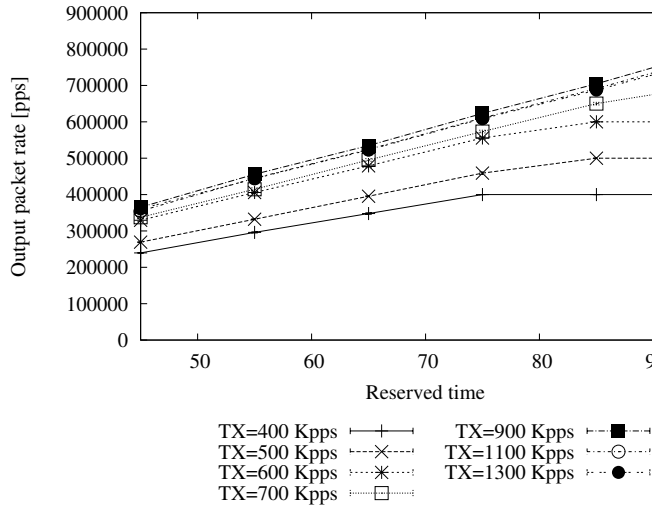


Figure 10: Performance of a monolithic VSR as a function of the amount of time reserved to the `vhost-net` kernel thread, when `vcpu` and `vhost-net` execute on 2 different cores.

The effectiveness of `SCHED_DEADLINE` in controlling the VSR performance has been studied through a large number of experiments, binding the `vcpu` thread and the `vhost-net` kernel thread to various CPU cores, and scheduling them through the CBS algorithm.

Fig. 10 plots the throughput achieved for different input rates when the `vhost-net` kernel thread and the `vcpu` thread are scheduled on different CPU cores, and the `vhost-net` kernel thread is scheduled by a CBS with $T^s = 200ms$ and Q^s ranging from $70ms$ (35% of the core's time) to $190ms$ (95% of the core's time). Throughput increases almost linearly with the amount of time reserved for the `vhost-net` kernel (until it reaches the peak forwarding rate).

Fig. 11 plots the results of a similar experiment in which the `vcpu` thread (instead of the `vhost-net` kernel thread) is scheduled using the CBS. Virtual routing performance increases almost linearly with the amount of reserved time, until the peak forwarding rate is reached. By comparing Fig. 10 and Fig. 11, it is apparent that the `vhost-net` kernel thread consumes more CPU time than the `vcpu` thread (the VSR reaches the maximum performance for larger values of the reserved CPU time).

Finally, Fig. 12 shows performance when the `vhost-net` kernel thread and the `vcpu` thread are scheduled on the same CPU core, and the `vcpu` thread is scheduled by a CBS with $T^s = 200ms$ and Q^s ranging from $70ms$ (35% of the core's time) to $190ms$ (95% of the core's time). Once again the throughput increases almost linearly with the amount of time reserved to the `vcpu` thread. However, in this case the peak throughput is lower, due to the fact that the two threads are sharing the same CPU core.

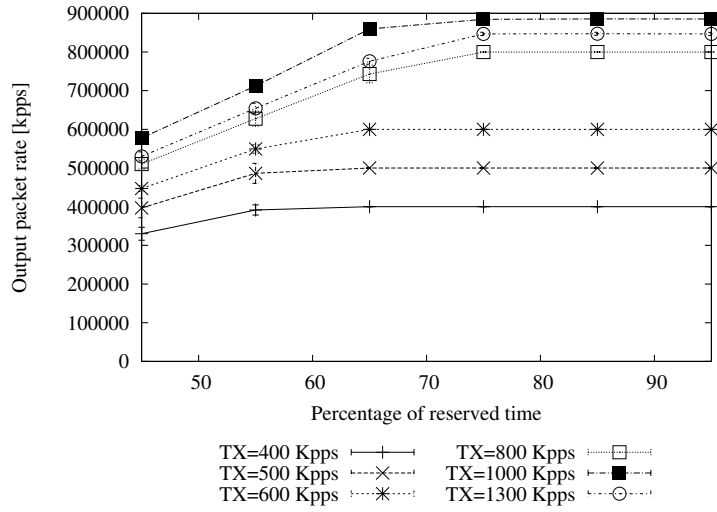


Figure 11: Performance of a monolithic VSR as a function of the amount of time reserved to the vcpu thread, when vcpu and vhost-net execute on 2 different cores.

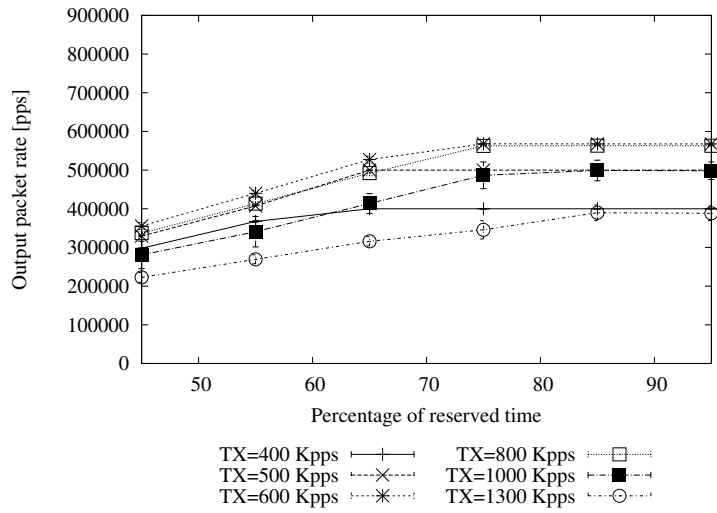


Figure 12: Performance of a monolithic VSR as a function of the amount of time reserved to the vcpu thread, when vcpu and vhost-net execute on the same core.

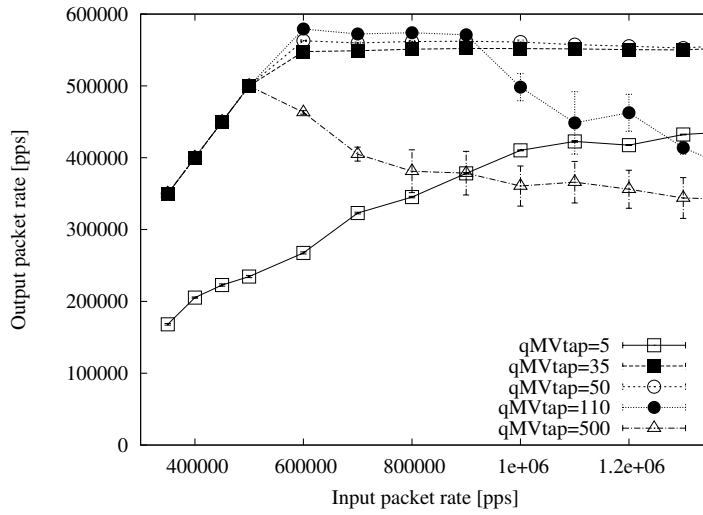


Figure 13: Performance of a monolithic VSR as a function of the macvtap queue size, when vcpu and vhost-net execute on the same core and are scheduled with real-time priorities.

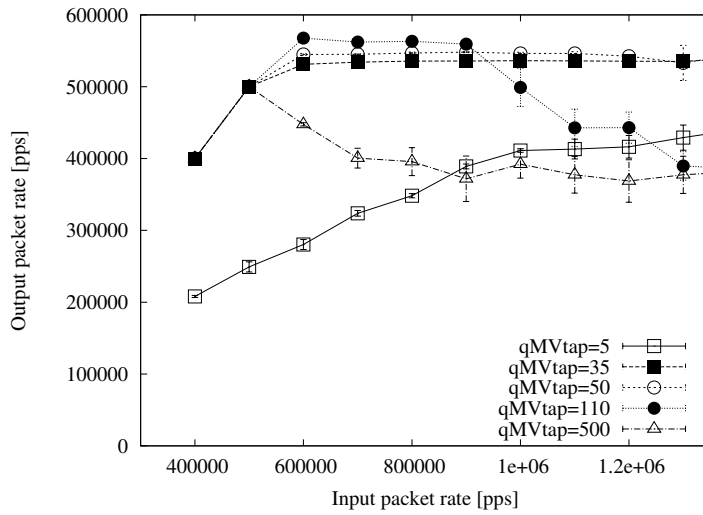


Figure 14: Performance of a monolithic VSR as a function of the macvtap queue size, when vcpu and vhost-net execute on the same core and the vcpu thread is reserved 170ms every 200ms.

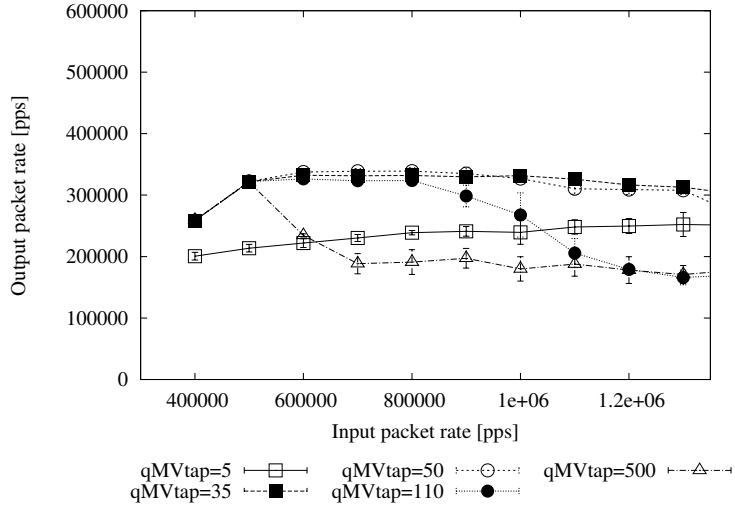


Figure 15: Performance of a monolithic VSR as a function of the `macvtap` queue size, when `vcpu` and `vhost-net` execute on the same core and the `vcpu` thread is reserved $90ms$ every $200ms$.

3.6. Effects of the `macvtap` queue size

As explained in Sec. 2 and in the Appendix, the packets received on the physical interface are queued in a `macvtap` (or TAP) device before being moved into the virtio ring by the `vhost-net` kernel thread. Some experiments on a monolithic VSR showed that when the VSR is overloaded, the size of this device’s queue can heavily affect the forwarding performance. This happens because a longer queue allows the `vhost-net` kernel thread to move more packets to the virtio ring, increasing the `vcpu` thread load and the number of packets sent back by the guest. Hence, the `vhost-net` kernel thread spends more time to move packets back to the `tx_ring` of the physical interface.

This seems to suggest that early packet dropping can avoid congestions in the `vcpu` thread and in the tx path. Hence, it can be conjectured that when the system is overloaded, the `macvtap` queue should be shorter to drop packets as soon as possible and to avoid spending CPU time in processing packets that will be dropped later. This intuition is confirmed by Fig. 13, showing the impact of the `macvtap` queue size on the performance of a monolithic VSR with the `vcpu` thread and `vhost-net` kernel thread scheduled on the same core. A queue size of 110 packets permits to reach a slightly higher peak rate, but provides a worse behaviour in overload. Smaller queue sizes (such as 50 packets or 35 packets) represent a good trade-off allowing to reach a slightly worse peak rate but showing no degradation in overload (thanks to the early packets dropping). Extreme values, such as 500 or 5 tend to provide worse performance. In particular, large queues behave well until the VSR is overloaded but are then subject to a huge performance degradation. Very short queues, on the other hand, perform badly at low loads.

Similar experiments were repeated when scheduling the `vcpu` thread with the CBS scheduling, and

reserving a variable amount of time for it. When the `vcpu` thread is reserved a large amount of time, the results are similar to the ones shown in Fig. 13. In particular, Fig. 14 shows the virtual routing performance obtained when the `vcpu` thread is scheduled by using a $(170ms, 200ms)$ CBS. Performance obtained for queue sizes of 35 and 50 packets are very similar to the previous ones, while the performance obtained for large queue sizes (500 packets) are slightly better in overload. This is due to the temporal protection mechanism provided by the CBS. Finally, Fig. 15 shows what happens in conditions of huge overload. The `vcpu` thread is controlled by a $(90ms, 200ms)$ CBS, hence it has not enough CPU time to route the packets.

4. Aggregation of Multiple Virtual Routers

As shown in Fig. 2, even when the host scheduler is correctly configured, a “monolithic” VSR (that is, a VSR based on a single VM running a SR) is not able to forward more than $900kpps$ when using small packets. Since the bottleneck lies in the `vhost-net` kernel thread, which consumes 100% of the CPU time of the core where it is bound, virtualising a multiprocessor machine does not improve performance (see Sec. 3.3). A solution to this issue could be to increase the number of `vhost-net` threads, by modifying the host and VMM setup and the mechanism used to move packets between physical and virtual interfaces. An alternative approach could be to modify the VSR structure, moving from a monolithic VSR to a routing architecture based on the aggregation of multiple software modules running inside multiple VMs. Such an aggregation can be performed using several different virtual routing architectures, exhibiting different characteristics in terms of performance, scalability and flexibility. In the following two different architectures are discussed.

The first example of such an aggregation is the Multistage Software Router (MSR) [3], which is based on a very flexible and feature-rich architecture, shown in Fig. 16.

A Multistage Software Router is composed by the following 3 stages:

- the first stage is composed by layer-2 *Load Balancers* (LBs) that distribute the input traffic load to some *Back-end Routers* (BRs);
- the second stage is the interconnection network. This is a mesh-based switched network between the first stage LBs and the third stage BRs. Multiple paths between the LBs and BRs could exist to support fault recovery;
- the third stage is composed by the BRs, i.e, forwarding engines that route packets to the proper LB.

A virtual Control Processor (Virtual CP) is used to coordinate the architecture as well as to unify the BRs’ routing table. The MSR hides its internal architecture and presents itself to external devices as a single router. As shown in previous work, the MSR architecture, when implemented on a cluster of physical machines, provides several interesting features, such as extending the number of interfaces one PC can host

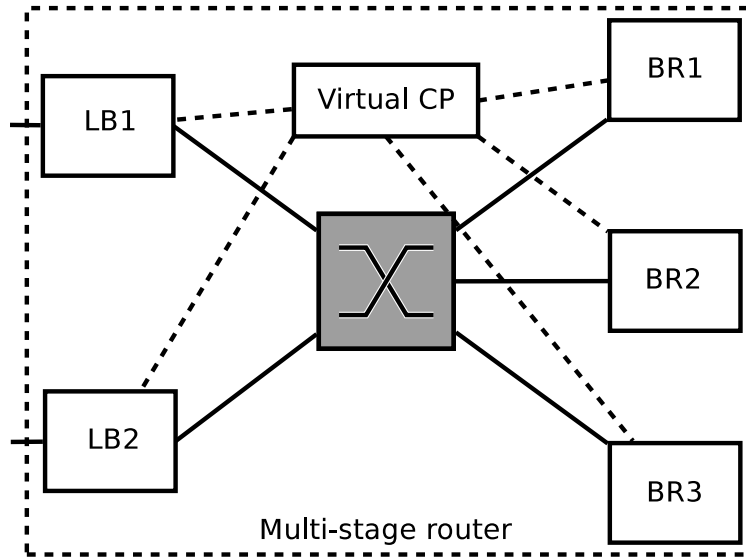


Figure 16: Example of the multi-stage router composed by two load balancers and three back-end routers.

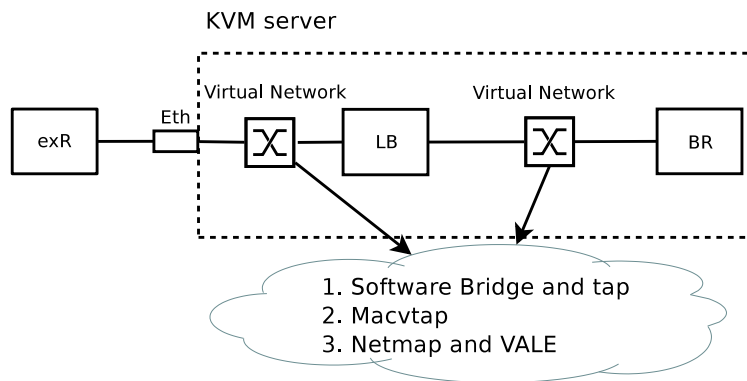


Figure 17: The implementation of the multistage software router inside a KVM server: 1 load balancer and 1 back-end router, with different interconnection networks.

(limited by the number of PCIe slots), dynamically shutting down unnecessary BRs at low traffic load while turning on BRs at high load, and seamlessly increasing the overall routing performance when necessary. Furthermore, it has been shown that, if LBs are implemented using FPGA hardware, the MSR's forwarding speed can scale almost linearly with the number of BRs. Instead, a software LB implementation based on Click modular router [17] permits to easily virtualise a MSR architecture using VMs for BRs and LBs.

The interconnection network is implemented inside the host as shown in Fig. 17, using openvswitch or a standard Linux software `bridge` and some pairs of TAP interfaces (since Figure 8 shows that openvswitch easily outperforms the standard bridge, it has been used in the experiments presented in this paper). Connection with the physical interface can be implemented using the Linux `macvtap` feature to improve performance when less CPU cores are available, as shown before.

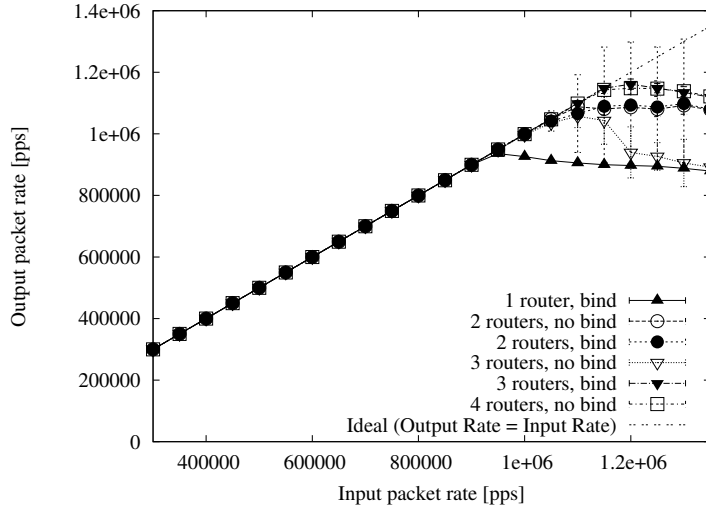


Figure 18: Performance of a PVR on 4 CPU cores, increasing the number of aggregated routers.

Implementing the first stage (LBs) with Click provides high flexibility: it become possible to build MSRs with a variable numbers of LBs and BRs, with a wide range of interconnection networks allowing for BRs distributions on different hosts, redundancy/fault tolerance, etc. However, it comes at the cost of consuming a huge amount of CPU time in the `vcpu` threads of the LBs, and in their `vhost-net` kernel threads. This means that the number of CPU cores needed to provide high performance becomes extremely high due to the software LB implementation.

If the focus is on forwarding performance (and some features/flexibility can be traded for higher performance), then a different VSR aggregation strategy can be used. The Linux `macvtap` interface provides a multi-queue functionality that can be used for load balancing: a single `macvtap` interface can split the traffic on multiple queues (currently based on network flows, but can be modified to distribute packets in a round-robin fashion). Such packet queues can be used by a single VM (using a multi-queue virtual network interface), or by multiple VMs. In this paper, this feature is used for running multiple identical copies of the same VSR, each one using a different `macvtap` queue. All VSRs run in identical VMs (having the same number of Ethernet interfaces, with the same IP and MAC addresses) and are seen from outside as a single VSR. Hence, the multi-queue `macvtap` aggregates all VSRs in a single VSR with the same configuration. This architecture, referred as Parallel Virtual Routers (PVR) architecture in this paper, is less flexible than MSR, but it removes the LB performance bottleneck (as shown in the next section).

5. Performance of Modular VSR

The next experiment focuses on the analysis of PVR performance. Fig. 18 displays how the performance of a PVR using the multi-queue `macvtap` mechanism for load balancing is affected by the number of aggre-

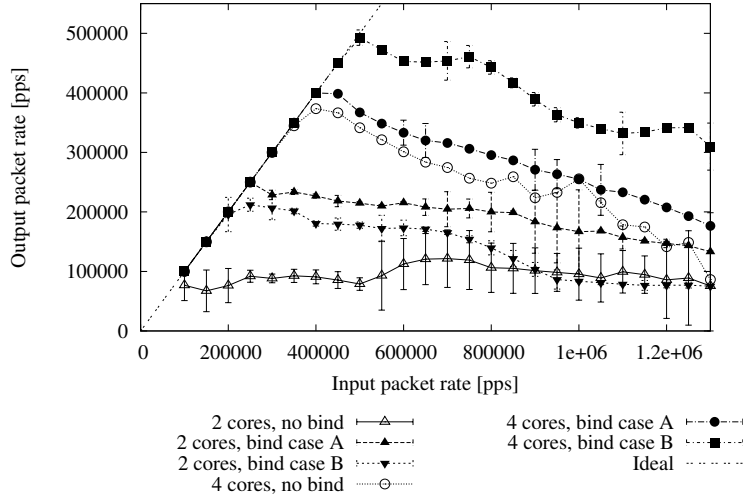


Figure 19: Effects of CPU bindings on the MSR performance.

gated routers. 4 CPU cores are used, and the setup is the same as the one used in Fig. 2 (same CPU, same number of runs per experiment, and the 99% confidence interval is displayed). This modular VSR is able to outperform a monolithic VSR (the best curve from Fig. 2 is repeated in Fig. 18 as “1 router, bind”) and reaches a forwarding rate of more than $1100kpps$. Furthermore, when aggregating 2 routers, CPU bindings have marginal impact on forwarding performance. On the other hand, when aggregating 3 routers, using proper bindings permits to better exploit the CPU time. Bindings become less relevant when increasing the number of routers, but performance do not improve, indicating that 4 CPU cores are not able to forward more than $1200kpps$ in a virtual architecture.

The last set of experiments focuses on analysing virtual MSR performance. Fig. 19 shows how the number of available CPU cores and the usage of correct CPU bindings affect the performance of a MSR. For the sake of simplicity and to easily understand the results, the figure refers to a simple MSR configuration with only 1 LB and 1 BR, but similar experiments with more complex setups were also performed providing results consistent with the ones presented here. Before analysing the results, consider that this MSR configuration (1 LB and 1 BR) creates 5 CPU-consuming threads: 1 `vcpu` thread and 2 `vhost-net` threads for the LB (since the LB has 2 virtual Ethernet interfaces), plus 1 `vcpu` thread and 1 `vhost-net` thread for the BR. As a result, the performance when executing on a single CPU core are pretty bad (the 5 threads easily overload a single core) and have not been reported in the figure.

When increasing the number of CPU cores to 2, the performance plot starts to become interesting, and appears to be quite sensitive to the CPU bindings: when no bindings are used, the MSR can route a little bit more than $100kpps$. In overload, the 5 threads tend to overload the CPU and keep bouncing between the 2 available cores. As a result, the variance in the forwarded throughput is higher (see the confidence interval).

Binding the `vcpu` threads of the LB and of the BR to the first core, and binding all the `vhost-net` kernel threads to the second core (“bind case A” in the figure) permits to achieve a higher maximum throughput (about $250kpps$). However, when the router is overloaded, throughput decreases (arriving to about $120kpps$ for an input rate of $1400kpps$). This happens because the three `vhost-net` threads overload the second core, when there is still some idle time on the first one. However, distributing the threads between cores in a different way (for example, see “bind case B” in the figure: the `vhost-net` kernel threads of the LB execute on the first core, together with the `vcpu` thread of the BR, all other threads are on the second core), does not seem to help and leads to a lower maximum throughput (about $200kpps$) and a worse behaviour in overload.

Finally, when increasing the number of cores to 4, the MSR performance further improves (because more CPU time is available for the 5 threads). Also in this case proper CPU bindings permit to improve the performance. The confidence intervals of the “4 cores, no bind” case are so large (reaching a size of more than $500kpps$) that have been removed from the plot for the sake of readability. By looking at the scheduler statistics and at the CPU usage inside the host, it is possible to see that the main performance problems are due to the `vcpu` thread of the LB, running Click, that consumes 100% of the CPU time on a core. This explains why the PVR architecture (which does not use Click for load balancing) is able to better exploit the computational power provided by the 4 CPU cores.

6. Related Work

The virtualisation of network resources is considered as a major opportunity to foster the innovation in the Internet and to solve the Internet *ossification* [18, 19] problem. Thus, many research projects are working on virtualisation technologies today. For instance, GENI [20] is a NSF research project (U.S.A.) on future Internet architectures where virtualisation technologies are used to create slices of network. On the European side, the FP7 FEDERICA project [21, 6] uses a similar approach based on virtualisation to create network slices to support research activities. Furthermore, network device manufacturers like Cisco and Juniper are supporting network virtualisation in their products as well. More recently, many companies like Amazon, Google and Microsoft are building services related to network virtualisation and cloud computing.

Researchers focused the attention on how to virtualise the key network component, i.e, routers or switches, and on performance improvements. Software routers based on XEN have been studied extensively [11, 22, 23]. The main conclusion is that the DomU, i.e, the guest domain, is not suitable for routing purpose due to the high overhead and low performance. More recent works on XEN [24, 25] show that it is possible to optimise the router internal architecture (i.e, maintain the packet in the same CPU cache with the help from multi-queue network interface support) to highly improve the SR’s performance. A different approach to virtualisation can be taken by using a container-base approach such as OpenVZ or LXC,

which can reduce the VM overhead. The implementation of VSRs based on such an approach has also been considered in literature [26]. Comparing those results with the ones presented in this paper, it is possible to notice that a proper configuration allows KVM to reach performance comparable with a container-based approach (or even better, but the comparison can be unfair because the experiments are based on different hardware).

Since the traditional Linux networking stack is designed for general purpose usage, some research works focused on optimising it for routing, or proposed alternative networking architectures to improve SR's performance. Netmap [2] and PF_RING [27] modify the NIC's drivers to directly map the NIC's ring buffers in user-space, bypassing the host networking stack to improve the performance. This approach is evaluated in Section 3.4, showing how it allows to achieve good performance with a small CPU usage, but it is not able to fully exploit all the available CPU cores. Packetshader [1] redirects received packets to the GPU for processing and forwarding. Experimental results show that Packetshader can improve significantly the throughput of CPU intensive workloads like encryption. Routebricks [4] exploits server clusters and load balancing to build a flat router architecture, similarly to the MSR solution.

Recently, multi-core architectures became very affordable, thus ways to parallelise processing and use all available resources to speed up SR performance became an appealing research topic. Several new technologies were introduced to improve networking performance, including Receive Side Scaling (RSS), Virtual Machine Device Queues (VMDq) and Extended Message-Signalled Interrupts (MSI-X), that are all technologies intended to utilise the multi-core server architecture efficiently [28]. The main idea is to spread the traffic load and network interrupts onto multiple cores while minimising cache misses at the same time. In the research side, some interesting works can be found as well. The performance impact of these mechanisms has been evaluated in various papers, for example by running software routers on a machine with multi-core CPUs and multi-queue NICs [29], or by studying the RSS and interrupt throttling mechanisms in XEN to improve the virtual software router performance [30]. Other works focused on evaluating the SR performance in multi-core systems by exploiting the standard Linux forwarding path [31].

7. Conclusions

This paper investigated the effects of some implementation and configuration details (scheduling affinity and priority, mechanisms used to move packets between VMs and physical interfaces, software router structure, ...) on the forwarding performance of virtual software router running in KVM. In particular, the paper showed:

- how to exploit the computing power provided by multiple CPU cores, by properly setting the CPU affinity of the various virtualisation activities;

- how to further improve the virtual forwarding performance by splitting the virtualisation activities into multiple threads;
- how a reservation-based scheduler allows to control the amount of CPU time consumed by a virtual router (and hence the VR performance);
- how the mechanisms used to move packets between VMs can affect the VR performance.

Moreover, it has been shown that a modular router architecture can help in better exploiting the computational power of the host. Two different architectures based on virtual routers aggregation have been analysed, optimising one of them for flexibility and the other one for performance (experimental results show that it is possible to outperform the monolithic architecture). Finally, the loose coupled aggregated software router architecture can be hosted in multiple servers, that can further boost the performance with more available resource as in the cloud computing scenario.

Acknowledgments

This research work is funded by the Italian Ministry of Research and Education through the PRIN SFINGI (SoFtware routers to Improve Next Generation Internet) project.

References

- [1] S. Han, K. Jang, K. Park, S. Moon, PacketShader: a GPU-accelerated software router, *SIGCOMM Comput. Commun. Rev.* 40 (4) (2010) 195–206.
- [2] L. Rizzo, Revisiting Network I/O APIs: The netmap Framework, *Queue* 10 (1) (2012) 30:30–30:39. doi:10.1145/2090147.2103536. URL <http://doi.acm.org/10.1145/2090147.2103536>
- [3] A. Bianco, J. M. Finochietto, M. Mellia, F. Neri, G. Galante, Multistage switching architectures for software routers, *IEEE Network* 21 (4) (2007) 15–21. doi:10.1109/MNET.2007.386465.
- [4] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, S. Ratnasamy, RouteBricks: exploiting parallelism to scale software routers, in: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09*, ACM, New York, NY, USA, 2009, pp. 15–28. doi:10.1145/1629575.1629578. URL <http://doi.acm.org/10.1145/1629575.1629578>
- [5] R. Bolla, R. Bruschi, G. Lamanna, A. Ranieri, DROP: An Open-Source Project towards Distributed SW Router Architectures, in: *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE, 2009*, pp. 1–6. doi:10.1109/GLOCOM.2009.5425461.
- [6] P. Szegedi, J. Riera, J. Garcia-Espin, M. Hidell, P. Sjodin, P. Soderman, M. Ruffini, D. O'Mahony, A. Bianco, L. Giraud, M. Ponce de Leon, G. Power, C. Cervello-Pastor, V. Lopez, S. Naegele-Jackson, Enabling future internet research: the FEDERICA case, *IEEE Commun. Mag.* 49 (7) (2011) 54–61. doi:10.1109/MCOM.2011.5936155.
- [7] M. Caesar, J. Rexford, Building bug-tolerant routers with virtualization, in: *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow (PRESTO08)*, Seattle, WA, USA, 2008.

- [8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, KVM: the Linux Virtual Machine Monitor, in: Proceedings of the Linux Symposium, Ottawa, Ontario, Canada, 2007.
- [9] L. Abeni, C. Kiraly, N. Li, A. Bianco, Tuning KVM to Enhance Virtual Routing Performance, in: Proceedings of the IEEE ICC'2013, IEEE, 2013, pp. 2396–2401.
- [10] A. Bianco, R. Birke, L. Giraudo, N. Li, Multistage Software Routers in a Virtual Environment, in: Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM 2010), Miami, Florida, 2010.
- [11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebar, I. Pratt, A. Warfield, Xen and the Art of Virtualization, in: Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP03), 2003.
- [12] F. Bellard, QEMU, a Fast and Portable Dynamic Translator, in: Proceedings of the 2005 USENIX Annual Technical Conference, Anaheim, CA, 2005.
- [13] R. Russel, virtio: Towards a De-Facto Standard for Virtual I/O Devices, ACM SIGOPS Operating Systems Review 42 (5).
- [14] A. Bianco, R. Birke, L. Giraudo, M. Palacin, OpenFlow Switching: Data Plane Performance, in: Communications (ICC), 2010 IEEE International Conference on, 2010, pp. 1–5. doi:10.1109/ICC.2010.5502016.
- [15] D. Faggioli, F. Checconi, M. Trimarchi, C. Scordino, An EDF scheduling class for the Linux kernel, in: Proceedings of the Eleventh Real-Time Linux Workshop, Dresden, Germany, 2009.
- [16] L. Abeni, G. Buttazzo, Integrating Multimedia Applications in Hard Real-Time Systems, in: Proceedings of the IEEE Real-Time Systems Symposium, Madrid, Spain, 1998.
- [17] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. F. Kaashoek, The Click modular router, ACM Trans. Comput. Syst. 18 (3) (2000) 263–297. doi:10.1145/354871.354874.
- [18] N. M. M. K. Chowdhury, R. Boutaba, Network virtualization: state of the art and research challenges, IEEE Communic. Mag. 47 (7) (2009) 20–26. doi:10.1109/MCOM.2009.5183468.
- [19] T. Anderson, L. Peterson, S. Shenker, J. Turner, Overcoming the Internet impasse through virtualization, Computer 38 (4) (2005) 34–41. doi:10.1109/MC.2005.136.
- [20] NSF GENI project, <http://www.geni.net/>.
- [21] The FEDERICA project, <http://www.fp7-federica.eu/>.
- [22] F. Anhalt, P. Primet, Analysis and experimental evaluation of data plane virtualization with Xen, in: ICNS, Valencia, Spain, 2009.
- [23] N. Egi, A. Greenhalgh, M. Handley, M. Hoerd, L. Mathy, T. Schooley, Evaluating Xen for Router Virtualization, in: ICCCN, Honolulu, HI, USA, 2007. doi:10.1109/ICCCN.2007.4317993.
- [24] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, S. Rixner, Achieving 10 Gb/s using safe and transparent network interface virtualization, in: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE '09, ACM, New York, NY, USA, 2009, pp. 61–70. doi:10.1145/1508293.1508303.
URL <http://doi.acm.org/10.1145/1508293.1508303>
- [25] N. Egi, A. Greenhalgh, M. Handley, M. Hoerd, F. Huici, L. Mathy, Towards high performance virtual routers on commodity hardware, in: Proceedings of the 2008 ACM CoNEXT Conference, CoNEXT '08, ACM, New York, NY, USA, 2008, pp. 20:1–20:12. doi:10.1145/1544012.1544032.
URL <http://doi.acm.org/10.1145/1544012.1544032>
- [26] S. Rathore, M. Hidell, P. Sjodin, Performance Evaluation of Open Virtual Routers, in: IEEE GLOBECOM, Miami, FL, USA, 2010.
- [27] NSF GENI project, http://www.ntop.org/products/pf_ring/.
- [28] Intel, Improving Network Performance in Multicore System, <http://www.intel.com/content/dam/doc/white-paper/improving-network-performance-in-multi-core-systems-paper.pdf>.
- [29] N. Egi, G. Iannaccone, M. Manesh, L. Mathy, S. Ratnasamy, Improved parallelism and scheduling in multi-core software

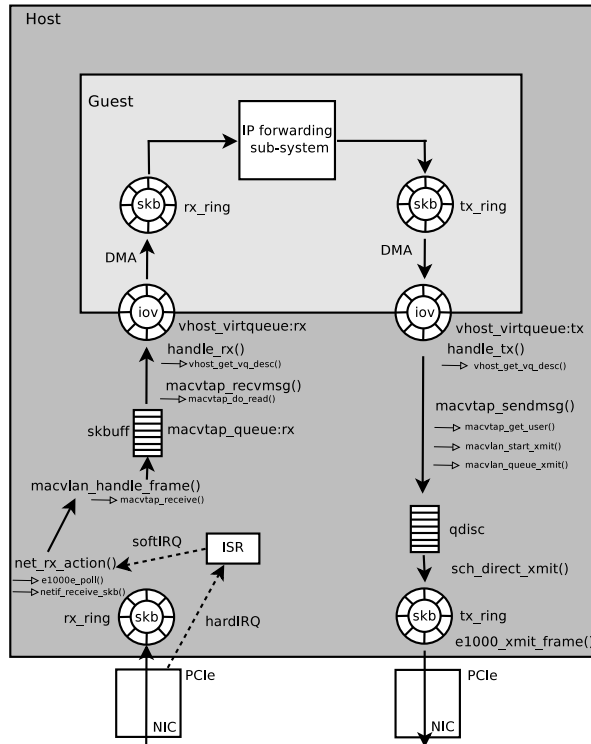


Figure A.20: The detailed scheme of packet path in virtualised software router architecture.

routers, *The Journal of Supercomputing* 63 (1) (2013) 294–322. doi:10.1007/s11227-011-0579-3.

URL <http://dx.doi.org/10.1007/s11227-011-0579-3>

- [30] H. Guan, Y. Dong, R. Ma, D. Xu, Y. Zhang, J. Li, Performance enhancement for network i/o virtualization with efficient interrupt coalescing and virtual receive-side scaling, *IEEE Transactions on Parallel and Distributed Systems* 26 (6) (2013) 1118–1128.
- [31] M. Dobrescu, K. Argyraki, S. Ratnasamy, Toward predictable performance in software packet-processing platforms, in: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12*, USENIX Association, Berkeley, CA, USA, 2012, pp. 11–11.
URL <http://dl.acm.org/citation.cfm?id=2228298.2228313>
- [32] J. H. Salim, R. Olsson, A. Kuznetsov, Beyond softnet, in: *Proceedings of the 5th annual Linux Showcase & Conference - Volume 5, ALS '01*, USENIX Association, Berkeley, CA, USA, 2001, pp. 18–18.
URL <http://dl.acm.org/citation.cfm?id=1268488.1268506>
- [33] R. Bolla, R. Bruschi, Linux software router: Data plane optimization and performance evaluation, *Journal of Networks* 2 (3) (2007) 6–17.

Appendix A. Moving Packets between Physical and Virtual Interfaces

A VSR is based on one or more VMs running in a host. The host has several physical Network Interface Cards (NICs) which receive/transmit packets to be delivered to/from the VMs. Hence, the host must

provide some mechanisms to move packets between physical and virtual NICs. Consider the most important operations performed for packet forwarding in a VSR:

- the physical NIC receives packets and stores them in a receive side data structure which is called `rx_ring`. The host is notified about packets arrival through some interrupts generated by the NIC;
- the network drivers (which are part of the host OS kernel) insert received packets in a software structure (`skbuff`), and the `vhost-net` kernel thread moves the `skbuffs` into a `virtio` ring (`vhost_virtqueue` in Fig. A.20) which is shared between host and guest. The guest is then notified (by triggering an interrupt in the guest) to do further processing;
- The frontend `virtio` driver inside the guest fetches the packets from the `virtio` ring and forwards them from the input virtual NIC to the destination one by running some forwarding code (for example, the Linux OS kernel in the guest);
- The guest OS notifies the host (activating the `vhost-net` kernel thread) that some packets are ready to be sent, and the `vhost-net` kernel thread delivers them to the correct physical NIC (by using the device driver, which puts the packets in a transmission side data structure called `tx_ring`);
- The transmission interface sends packets out and cleans/recycles the resource.

The details of the packet life cycle in the VSR is described in Fig. A.20. The NICs are connected to PCIe slots, and during the boot of the host OS, multiple memory regions, corresponding to the so called `tx_rings` and `rx_rings`, are allocated to NICs. These data structures are located in the main memory (i.e, RAM) and can be accessed by both the CPUs and the NICs (modern NICs use *bus mastering* to move data between the rings and the physical interface). As soon as a packet is received by a NIC, it is stored in the `rx_ring` by the NIC (using bus mastering), and a hardware interrupt is eventually generated to notify the kernel for processing. The kernel handles such an interrupt by running an Interrupt Service Routine (ISR), or `hardIRQ` handler, which sends a fast acknowledgement to the NIC. The packet, however, is not processed in `hardIRQ` context, but packet processing is deferred to a later handler (the so called `softIRQ` handler), which can perform more time consuming operations. When the packet arrival rate is very high, the excessive number of generated interrupts (or the resulting interrupt storms) could introduce too much overhead and decreases the system throughput significantly. This issue is addressed by some interrupt mitigation mechanisms implemented in the hardware, and by the NAPI [32] mechanism implemented in the Linux kernel, which can reduce the activation rate of `hardIRQ` handlers by adaptively using a polling technique. Notice that modern network cards/drivers enable NAPI by default, and we exploit this feature in both host and guest.

Indeed, when a hardware interrupt is received the NAPI driver just acknowledges the NIC and triggers a `softIRQ`. Such a `softIRQ` executes the `net_rx_action()` function, which calls the driver to receive the packets from the `rx_ring`, and can start polling the NIC (disabling the generation of hardware interrupts) if the packet arrival rate is too high and risks cause a high interrupt load. Polling will stop (and hardware interrupts will be re-enabled) as soon as a lower packet arrival rate is detected.

Many solutions exist for moving packets into the guest, by queueing them in some kind of device, and using some mechanism to move packets from such a device to the guest. For instance, a software bridge could be created by running the Linux `brctl` utility, thus a layer 2 network between the host and the guest is built. In this solution, a TAP interface is used to make the packets accessible to the VMM. Alternatively, a different and more efficient device - the `macvtap` device - can be used. In this case, the `macvtap` interface is directly bound to the physical NIC (performing a MAC-based filtering) and can be directly accessed by the VMM to improve performance. More recently, the `netmap` solution, which is based on directly mapping the NIC rings in user space to make them directly accessible by the VMM, has been proposed for connecting the guest with host or directly with a NIC. Sec. 3.4 presents a detailed comparison of these 3 approaches. Here for the sake of simplicity we focus on the description of `macvtap`.

As shown in Fig. A.20, the driver (invoked by `net_rx_action()`) receives the packets and passes them to `netif_receive_skb()`, which in turn passes them to the `macvtap` device (without passing through the host's network protocol stack). Then, packets are enqueued in the `macvtap` interface from which packets are moved to the `virtio` ring by the `vhost-net` kernel thread. This action is performed by the `macvtap_receive` function, which is invoked by `macvlan_handle_frame()` when a packet is received, and inserts such a packet in queue contained in the `macvtap` interface. Then, the `vhost-net` kernel thread polls packets from the `macvtap` queue by calling the `handle_rx()` function and after processing, it eventually inserts the packets into a shared memory region (called `vhost_virtqueue`). The `vhost-net` kernel thread notifies the user space guest of the arrived packet events by registering them into the KVM kernel module, which has the feature to trigger the corresponding guest rx interrupts for further processing.

The guest is then notified of the received packet, and processes it according to the traditional Linux forwarding schema as described in [33]. Roughly speaking, the guest kernel fetches packets from `vhost_virtqueue` to the `rx_ring` inside the guest, with the help of the `virtio` frontend driver. Packets are then passed to the IP forwarding sub-system of a standard Linux kernel and, after processing, i.e, header checksum, TTL decrease, destination matching operations etc, they are sent out to the correct virtual interface and placed into the corresponding `vhost_virtqueue`. The guest then kicks the `vhost-net` kernel thread by means of a registered I/O event file descriptor, which the `vhost-net` thread monitors.

Packets are then moved back to a physical NIC by the host kernel:

- packets are moved from the `vhost_virtqueue` into the `qdisc` by calling the `handle_tx()` and `macvtap_sendmsg()`;

- packets transmission is scheduled based on the QoS rules defined to the queueing discipline (`qdisc`) configured in the host, by calling `sch_direct_xmit()`;
- the NIC driver (i.e, the `e1000_xmit_frame()` function as shown in Fig. A.20) sends them to the wire;
- the memory is cleared and recycled for future use.

The specific function calls depend on the specific drivers used by the host and on the mechanism used to move packets between host and guest, but the path followed by packets in VSR is similar in all solutions.