Data Plane Optimization in Open Virtual Routers

M. Siraj Rathore, Markus Hidell, Peter Sjödin

Telecommunication Systems (TS) Lab, School of ICT, KTH-Royal Institute of Technology,
16440 Kista, Sweden
{siraj, mahidell, psj}@kth.se

Abstract. A major challenge in network virtualization is to virtualize the components constituting the network, in particular the routers. In the work presented here, we focus on how to use open source Linux software in combination with commodity hardware to build open virtual routers. A general approach in open router virtualization is to run multiple virtual instances in parallel on the same PC hardware. This means that virtual components are combined in the router's data plane, which can result in performance penalty. In this paper, we investigate the impact of the design of virtual network devices on router performance in Linux namespace environment. We identify performance bottlenecks along the packet data path. We suggest design changes to improve performance. In particular, we investigate modifications of the "macvlan" device, and analyze the performance improvements in terms of packet forwarding. We also investigate how the number of virtual routers and virtual devices within a physical machine influence performance.
Keywords: network virtualization, virtual router, SoftIRQ, NAPI, Softnet API.

1 Introduction

With the continuous growth of the Internet to new areas, services and applications, the demands increase on the ways in which we organize and manage networks. Future networks need to be flexible, support a diversity of services and applications, and should be easy to manage and maintain. One way of addressing these requirements is to *virtualize* routers. Router virtualization involves running several router instances on the same physical hardware, in a way that allows each instance to appear as a separate, independent router. This makes it possible to support a multitude of services, management disciplines, and protocols in parallel on different virtual routers.

A general approach to router virtualization is to use computing virtualization techniques to run multiple operating systems as guests in parallel on the same hardware, and let each guest run one instance of the router software. In *open source virtual routers*, these operating systems are based on open source software that can be combined with commodity PC hardware.

Broadly virtualization techniques can be categorized into hypervisor-based and container-based virtualization. With hypervisor-based virtualization (such as KVM and Xen), the hardware is virtualized so that each guest runs its own operating system. In contained-based virtualization (sometimes also called jails or operating system virtualization), the operating system is partitioned into multiple domains, where each guest runs in its own domain. Examples of container-based virtualization include FreeBSD Jail, Linux-VServer, OpenVZ, and Linux Namespaces. Container-based virtualization is more "light weight" in the sense that it is based on partitioning of the operating system resources, something that could potentially be achieved with no or little extra processing overhead. The drawback is that all guests need to run the same operating system, as they are sharing the same operating system kernel. Hypervisor-based virtualization is more flexible in this respect but incurs a larger overhead, which limits the number of concurrent guests.

The purpose of our work is to study how container-based virtualization can be used for router virtualization. We focus on contained-based virtualization because of its potential for low processing overhead and support for many simultaneous guests. The starting point for our work is the observation that there is a significant performance penalty for performing packet processing in guests. The performance penalty comes from the level of indirection between network interfaces and virtual routers: When a packet arrives on a network interface, the corresponding virtual router should be identified, and the packet should be redirected to the virtual router. The way in which this redirection is performed has fundamental impact on overall performance.

A common solution is to use existing kernel components such as software bridges and virtual interfaces. In previous work we have shown that this is a costly solution in terms of performance, as it introduces considerable processing overhead [9]. A more promising approach is to use the *macvlan* device – a kernel object specifically designed for support of multiplexing and demultiplexing of packets between physical and virtual interfaces based on MAC addresses. The purpose of this paper is to investigate how a virtual router framework can be designed around the macvlan device and Linux Namespaces. We find that macvlan exhibits undesirable behavior at overload – when traffic load increases above a certain point, the effective throughput goes down. This is a considerable disadvantage of any routing platform, and we therefore propose a revised version, called NAPI-macvlan. The performance of NAPI-macvlan is studied in terms of throughput, scalability, and virtual router isolation properties. We demonstrate that NAPI-macvlan has superior performance, compared to macvlan, and does not exhibit the negative behavior at overload.

The rest of this paper is organized as follows: Section 2 surveys related work on virtual router platforms. Thereafter, Section 3 describes a Linux virtual router framework based on macvlan devices and Namespaces. This section also introduces the NAPI-macvlan and gives the rationale behind its design. Section 4 presents and analyses performance measurements of virtual router configurations using macvlan and NAPI-macvlan. Finally, Section 5 concludes the paper.

2 Related Work

There are many examples of work, where different virtualization technologies are evaluated as virtual router platforms. For instance, Xen virtualization has been investigated in detail [1] and it has been shown that Xen can achieve a considerable forwarding rate in the privileged domain, but that guest domain forwarding results in poor performance. Studies on data plane virtualization using Xen can also be found

[2], where packet forwarding through guest domain is suggested in order to virtualize the data plane. Another work demonstrates how to make efficient use of multicore commodity hardware for virtual routers [3]. It also identifies performance bottlenecks associated with the currently available commodity hardware.

VINI [4] presents a virtual network infrastructure for network experimentation in a realistic and controlled environment. For router virtualization, VINI uses User-Mode Linux (UML) and Click to define custom data planes. The data plane runs in user space, something which gives great flexibility, but the implementation is also subject to a significant performance penalty.

Another platform for virtual networks is Trellis [5]. Router virtualization in Trellis is based on customized components that have been introduced to improve forwarding rates. The resulting throughput is compared with virtual routers based on Xen and Openvz, it is shown that higher throughput can be achieved in Trellis.

An alternative virtual router enabling technology is to use a source code merging scheme as a mechanism to define custom data planes for virtual routers [6]. Source code is a language used to define the packet path for each virtual router. The packet path is specified in terms of networking functions that are connected together. Click and Linux VServer is used to provide virtual routers platform.

PdP (Parallel data Plane) presents a virtual network platform to achieve high speed packet processing [7]. It runs control and data planes in guest machines for better isolation and flexibility. In order to boost the forwarding rate, PdP uses an architecture where multiple guest machines perform packet forwarding in parallel.

The Crossbrow architecture [8] is yet another example of a virtual network platform. The focus is network resource virtualization to achieve fair bandwidth sharing among various virtual instances. Virtualization of physical network interfaces is proposed using different virtual devices (such as VNIC and virtual switch).

3 Linux Based Virtual Routers

A virtual router sends and receives packets on virtual interfaces. Besides this, a virtual router is just like a physical router: is has a routing table, routing protocols, packet filtering rules, management interface, and so on. A virtual router runs in a host environment, which is responsible for allocating resources to the virtual router, and for managing these resources. There can be multiple virtual routers in the same host environment, sharing the available resources, as illustrated in **Fig. 1**. Even though a virtual router communicates on virtual interfaces, its purpose is often to process packets that appear on the physical interfaces in the host environment. This means that the host environment needs to redirect packets between physical and virtual interfaces. When a packet is received on a physical interface, the host environment checks the packet in order to identify the virtual router to which the packet should be redirected, and makes the packet available to the virtual router on one of its virtual interfaces, When the virtual router has processed the packet and determined the next hop, the packet is placed on an outgoing virtual interface from where it is finally transmitted on a physical interface.

The redirection of packets between physical and virtual interfaces introduces a layer of indirection that is not present in a physical router. This functionality can be implemented in several ways. One common configuration is to use the regular software bridge module in the Linux kernel to interconnect the interfaces, as shown in [9]. The software bridge provides a general-purpose switching function that allows packets to be switched between interfaces (virtual or physical) based on MAC addresses and MAC address learning. This solution is general in the sense that it allows packet to be switched between any pair of interfaces, and it is an attractive solution being based on a well-known software component already available in the kernel. However, for the purpose of redirecting packets between virtual and physical interfaces, it introduces a considerable amount of overhead. This, in turn, incurs performance penalties, something that was investigated in previous work [9]. Similar solutions that have been used are the virtual switch [14] and the short bridge [5]. It is also possible to use, for example, IP routing and Network Address Translation (NAT) for traffic to and from virtual machines, but those are not suitable for virtual routers.

A promising solution from a performance point of view is to replace the software bridge with a multiplexing/demultiplexing module. This is a more restricted solution, but potentially more efficient, since it can move packets from physical interface to virtual with less overhead. In the following we will investigate this solution closer. We start by examining the packet processing path in a Linux-based virtual router in more detail.

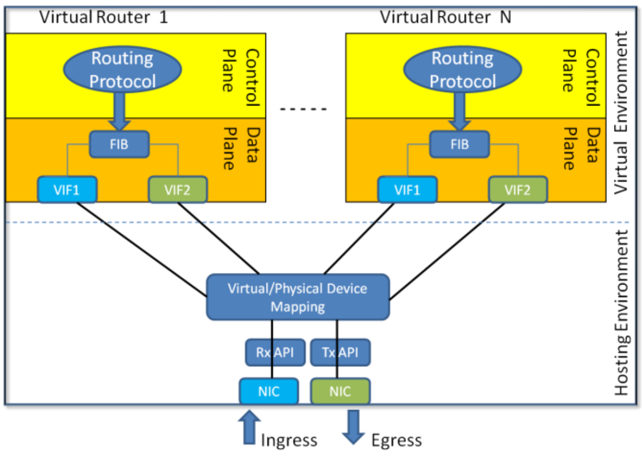

Fig. 1. General design of Linux-based virtual routers

3.1 Packet Processing Path

When a packet is received on a **Network Interface Card (NIC)**, the packet is transferred to a receive buffer in main memory, and an RX (receive) interrupt is generated. Interrupt processing is costly and can have large impact on forwarding performance [10]. Modern NICs provide features like interrupt coalescing and interrupt throttling for mitigating the negative effects of interrupt handling [11]. Current Linux use an RX interrupt processing scheme called NAPI [12], adopted from kernel 2.4.20 [13]. Instead of using one hardware interrupt per packet, NAPI combines interrupts with polling so that multiple packets can be processed within a single hardware interrupt. The NAPI interrupt handler accumulates packets in receive buffers, and schedules a software interrupt (SoftIRQ) to trigger processing of a batch of incoming packets.

After a SoftIRQ, the packet is delivered towards a virtual router via the corresponding virtual interface (VIF). Virtual interfaces are exactly like physical interfaces except that they are completely implemented in software. Therefore, no hardware interrupts are involved. A common example of a virtual interface is *veth*—the virtual Ethernet device. It has its own MAC address and an administrator has full access to its configuration in terms of MAC address, IP address assignment etc. Another example of virtual interfaces is the *macvlan* device, which will be discussed in more detail below. Both veth and macvlan are the part of the Linux kernel (2.6.x). When the virtual router has processed a packet from its incoming virtual interface, the packet will be scheduled for transmission on an outgoing NIC. The outgoing NIC has a transmit (TX) queue where outgoing packets are placed. The NIC is then informed that outgoing packets are ready for transmission. The NIC's DMA engine fetches the packet from host memory and transmits it onto the physical media.

3.2 The macvlan Virtual Interface

In previous work we investigated how a careful selection of virtual interface and approach for redirecting packets between virtual and physical interfaces can improve the overall system performance [9]. We compared a macvlan-based virtual router with veth/bridge-based virtual router using both OpenVZ and Linux namespaces virtualization environments. We concluded that, in comparison with veth/bridge, a macvlan based virtual router is far less CPU demanding and can achieve higher throughput. In addition, it shows better behavior in overload situations.

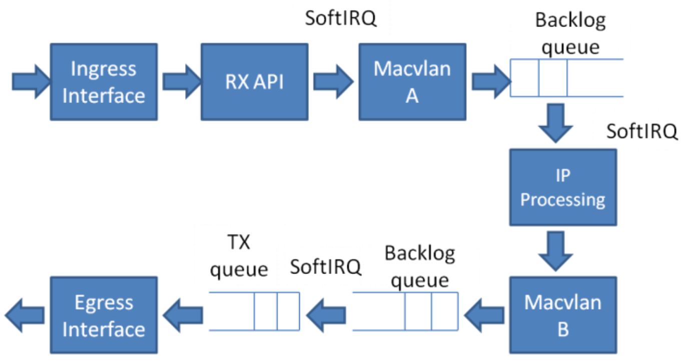

Fig. 2. Data plane for a macvlan based virtual router

The macvlan device provides a mechanism to define multiple virtual interfaces on top of a single physical interface. Each virtual interface is bound to a physical interface and has its own MAC address. A MAC address table is maintained for the virtual/physical address mapping. **Fig. 2** depicts the packet data path for a macvlan-based virtual router. A packet received in a physical ingress interface enters into the virtual router through a virtual interface, macvlan A. After MAC level processing, the packet is queued in a *backlog queue* and a SoftIRQ is scheduled. When the SoftIRQ occurs, the packet is fetched from the queue for processing in the virtual router. As a result of the processing, an outgoing virtual interface is determined, and the packet is handed over to this virtual interface, macvlan B. There, the packet is buffered in another backlog queue and a new SoftIRQ is scheduled. Finally, upon the new SoftIRQ, the packet is moved from the backlog queue to the transmission queue of the physical egress interface.

Like a physical interface, a virtual interface always delivers a packet for receive interrupt processing after interface level processing. It can be observed in **Fig. 2** that there is a backlog queue following each macvlan device. This design stems from the use of the Softnet API (a predecessor of NAPI). Upon completion of interface level processing, the macvlan device calls Softnet API, which buffers the packet in a backlog queue and schedules a SoftIRQ for further packet processing [12]. The Softnet API performance limitations are well known for physical interfaces [10], and can create livelock situations at high traffic loads. We believe that the use of the Softnet API for virtual interfaces has the following design weaknesses:

- Backlog queue congestion can cause serious throughput degradation for bridge/veth based virtual routers [9].
- The backlog queue is maintained on a per CPU basis, which means that virtual routers running on the same CPU will share the same queue. This may result in resource contention, something that can corrupt isolation properties between virtual routers. It may also limit overall system scalability.
- There are multiple queuing points along the data path. This may cause unnecessary delays in packet processing, and lead to inefficient usage of CPU resources.

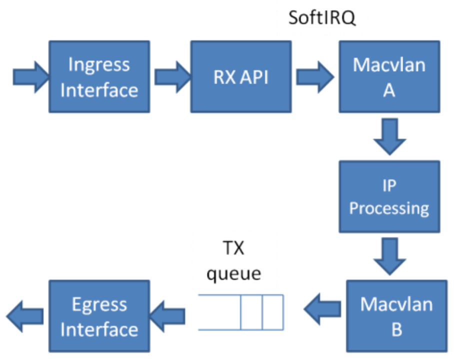

Fig. 3. Data plane for a NAPI-macvlan based virtual router

In order to address these issues, we introduce two changes in the data path:

- The backlog queues are eliminated from the data path.
- Packet receive processing is modified to that a packet is carried from the ingress interface, through virtual router, to the TX queue of outgoing physical interface in a single SoftIRQ.

The modified data path is illustrated in **Fig. 3**. We have implemented this data path as a modified macvlan device, called *NAPI-macvlan*. It is different from a macvlan device in the sense that it uses NAPI (netif_receive_skb) instead of Softnet API so that packets are delivered directly to the next processing module without being stored in an intermediate queue.

4 Performance Evaluation

This section presents a performance evaluation of macvlan and NAPI-macvlan based virtual router platforms from different performance perspectives, considering throughput, scalability, isolation and latency. In all test cases, we use Linux namespaces as the foundation for virtualization. We relate the performance of a virtual router to regular IP forwarding in a non-virtualized Linux based router. Throughout this section, the latter is denoted "IP Forwarder", and we use it as a reference to study the effects of applying virtualization.

4.1 Experimental Setup

We adopt a standard method to examine router performance in conformance with RFC 2544 [16]. A source machine generates network traffic that passes through a device under test (DUT), which forwards towards the destination machine. The nodes are connected using 1 Gb/s links.

We use an AMD Phenom quad core 3.2 GHz machine as traffic generator (GEN). The machine is equipped with 4GB of memory and two Gigabit Ethernet Intel® PRO/1000 PT server adapters (controller chip 82571GB). Another machine with the same specifications is used as destination (SINK). We use an Intel dual core 2.6 GHz machine (E8200) as the DUT. The machine has 4GB memory and a quad port Gigabit Ethernet Intel PRO/1000 PT server adapter. The DUT is running Linux kernel version 2.6.33-Netnext. All network interfaces are running NAPI-aware network drivers. Interrupt throttling is turned off for all interfaces. We use pktgen [17], an open source tool for traffic generation and throughput computation at sink. For all tests, 64 byte packets are generated. The DUT is running a single CPU, unless something else is specified.

4.2 Throughput

Case I

We start with a simple scenario: a DUT with two physical interfaces, forwards packets from one interface to another (unidirectional traffic flow). The results are show in **Fig. 4**. With this scenario, non-virtualized IP forwarding reaches a maximum throughput of 785 kpps. This provides the baseline forwarding performance as a reference for the other measurements. **Fig. 4** also shows the throughput for virtual routers with two virtual interfaces, using macvlan devices as well as NAPI-macvlan. One virtual interface is connected to the ingress physical interface while the other virtual interface is connected to the egress physical interface. **Fig. 4** shows that the macvlan-based virtual router attains up to 690 kpps while the the NAPI-macvlan-based virtual router achieves around 700 kpps.

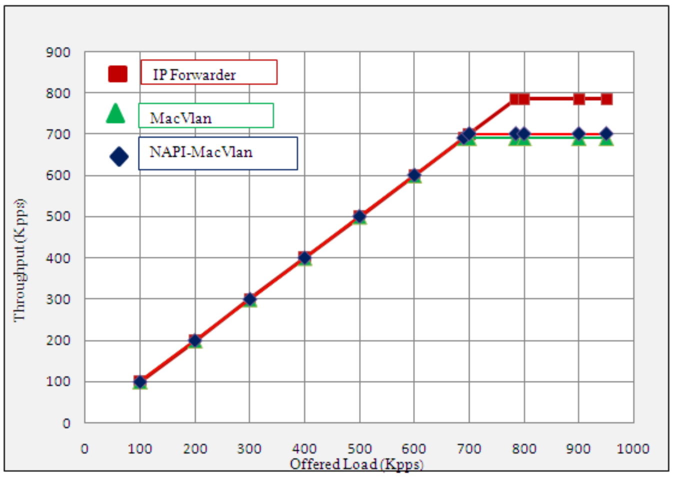

Fig. 4. Load vs throughput (Case I)

Case II

For the next step we add a physical interface to the setup. We offer load on two interfaces and use the third as egress interface. A single CPU core is thus processing traffic belonging to three physical interfaces. For this scenario, we consider aggregate offered load on the system (on both physical interfaces). The results are shown in **Fig. 5** (three interfaces). The native IP Forwarder reaches a packet rate of 785 kpps. The macvlan-based virtual router achieves a peak rate of 600 kpps at an offered load of around 600 kpps. Then, as the load increases, the packet rate will drop down to 480 kpps. The NAPI-macvlan-based virtual router peaks at 700 kpps, which is sustained as the offered load increases—a significant performance difference under overload.

To understand the throughput difference between macvlan and NAPI-macvlan, we study packet drop locations along the data path. In case of macvlan (**Fig. 2**), we find that packets are dropped at two locations: in the backlog queue after the incoming macvlan device (after “Macvlan A” in **Fig. 2**) and on the ingress physical interface. When the load is increased beyond 600 kpps, packet drop starts in the backlog queue. Increasing offered load results in more packet drop and throughput degradation, but packets are still accepted by the ingress interfaces. This situation remains until the aggregated load on the ingress interfaces reaches 1000 kpps. Above this load level, the ingress physical interface starts dropping packets, and from that point a throughput of 480 kpps is maintained for increasing load, as shown in **Fig. 5**. This type of behavior has been explained in detail for veth/bridge-based virtual routers [9]. In case of NAPI-macvlan, the only packet drop location is on the ingress physical interface. When the load is increased above 700 kpps (**Fig. 5**), the ingress interfaces start dropping packets, and a throughput of 700 kpps is sustained for higher loads. Clearly, this more graceful overload behavior is much more preferable.

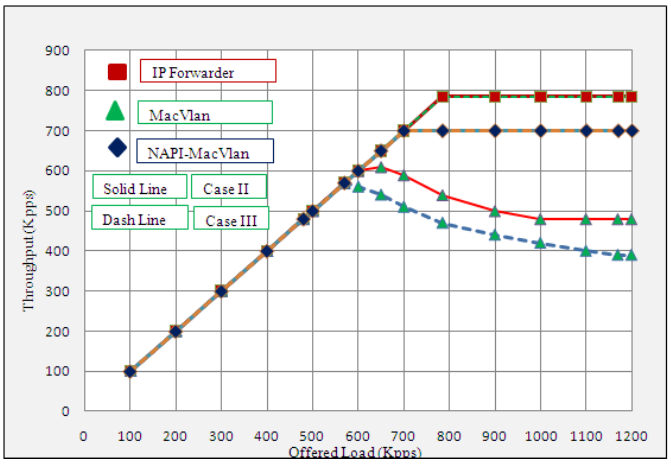

Fig. 5. Load vs throughput (II, III)

An interesting conclusion that can be made from this test case is that the cost of dropping packets in software is significant compared to dropping packets directly on the ingress interfaces. In Case II, we explain the gradual throughput degradation of macvlan virtualization for loads between 600 and 1000 kpps by the fact that packets are still accepted by the ingress interfaces and accordingly the CPU has to spend cycles both on packets that get dropped and on packets that get forwarded. Spending CPU cycles on packets that eventually will be dropped is clearly a waste of resources, adding overhead and reducing performance. This behavior cannot be seen for NAPI-macvlan virtualization since the only drop location is on the physical interface. Once a packet has been accepted by the macvlan device from the physical interface, the packet will be processed throughout the entire data path and delivered to the egress interface.

Case III

In this test case, we add yet another physical interface to the DUT and spread the offered load over three ingress interfaces while having a single egress interface. In this case, throughput for macvlan degrades even further. The backlog queue is now shared by even more ingress interfaces, something which increases the drop rate at the CPU level. The performance of NAPI-macvlan, on the other hand, is not affected by adding yet another physical ingress interface.

4.3 Scalability

Our first scalability study is to analyze the impact of running multiple virtual routers on the same physical platform. Therefore, we extend the experimental setup of case III with an increasing number of virtual routers. We create up to 32 virtual routers, each with four virtual interfaces (three ingress and one egress). Each ingress physical interface is now shared by up to 32 ingress virtual interfaces. A pktgen script is used to send 10 million packets in a row through one virtual router at a time. We do not observe any noticeable throughput degradation in the DUT when adding virtual routers. It can be seen in **Fig. 6** that the maximum throughput for the DUT remains almost constant irrespectively of the number of existing virtual routers (1 to 32). This behavior is the same both for macvlan and NAPI-macvlan. The tests indicate a promising scalability property of Linux namespaces. Another important scalability concern is to study the impact of IP route lookup on forwarding performance. Until now we consider a unidirectional virtual router with a single routing table entry—a valid test setup but not a practical scenario. We move towards a more realistic scenario by considering bidirectional traffic flows together with a larger routing table. We extend the setup of case I and update the virtual router with 512 routing table entries. We offer load on both physical interfaces, which is forwarded towards each other through the virtual router (i.e. bidirectional). As a first step, we offer load with the same IP destination in all packets. In this case, routing information is available in routing cache and there is no need to consult routing table.

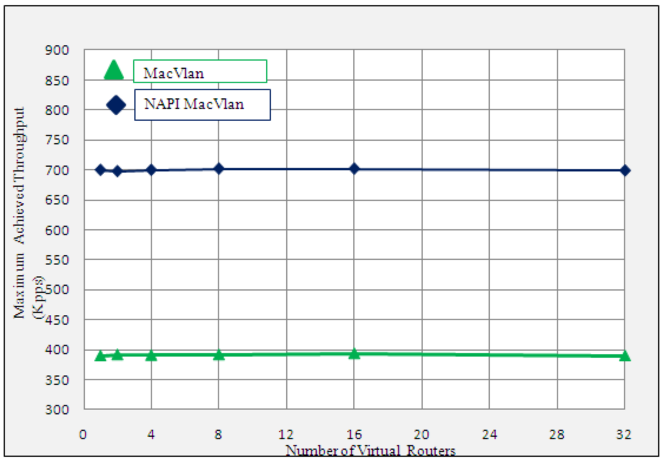

Fig. 6. Throughput vs. no. of virtual routers

The results are shown in **Fig. 7** (single destination, 2 virtual interfaces). The NAPI-macvlan router achieves 700 kkpps and macvlan router reaches 660 kpps. A minor drop rate in the backlog queue can be observed for macvlan. As the next step, we offer load with 512 different IP destinations to see the impact of having the CPU make a route lookup with misses in the route cache. **Fig. 7** (multiple destinations, 2 virtual interfaces) shows a minor throughput degradation for NAPI-macvlan (680 kpps). However, for macvlan packet drops in the backlog queue are now becoming substantial. As a result, throughput drops to 500 kpps. In the above scenario, the virtual router has only two virtual interfaces so there are only two entries for virtual/physical device mapping. All packets arrive on one interface and are transmitted on the other. Such a setup will not allow studies of the impact of virtual/physical device mapping, something which is important from scalability perspective. Furthermore, a virtual router with two virtual interfaces may not always be useful. We increase the number of virtual interfaces in the setup. For each physical interface, 256 virtual interfaces are created (512 in total). The bidirectional load is offered with 512 different IP destinations so that different egress virtual interfaces will be used for different destinations. We investigate the impact of device mapping.

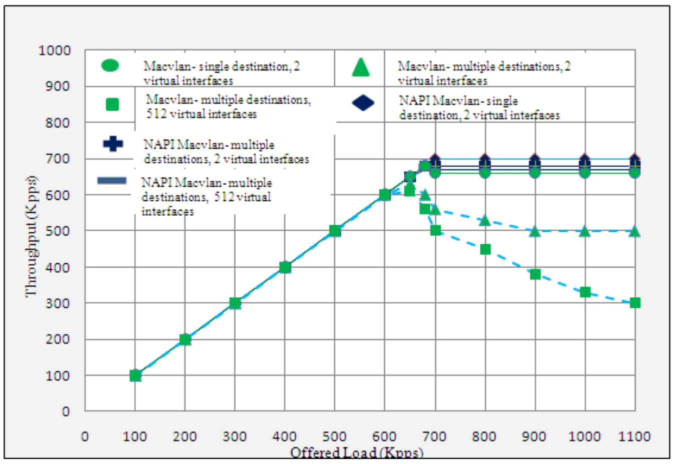

Fig. 7. Throughput for 2, 512 VIFs.

The results are shown in **Fig. 7** (512 virtual interfaces). It shows a small throughput degradation for NAPI-macvlan (670 kpps), compared to the earlier peak rate of 700 kpps. For macvlan, throughput degrades to 300 kpps as a result of more severe backlog congestion. We observe that when the computational burden increases on the CPU, the backlog congestion becomes more adverse.

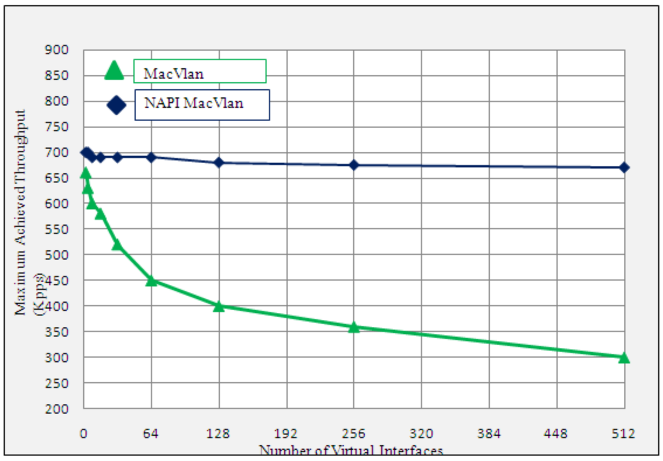

Fig. 8. Throughput vs VIFs.

Fig. 8, gives a more detailed picture of throughput versus the number of virtual interfaces. The number of IP destinations in the traffic flow is increased along with the number of virtual interfaces. The impact is minor for the NAPI-macvlan based router. However, throughput drops sharply for the macvlan case.

4.4 Isolation

In the previous subsection we verified that an increasing number of passive virtual routers had almost no effect on the overall throughput performance for neither macvlan nor NAPI-macvlan (**Fig. 6**). However, it is also important to investigate how multiple active virtual routers, running on the same CPU, might affect each other's operation. We refer to this as isolation properties. In this subsection, we study these isolation properties through an experiment where we analyze how a high offered load on one virtual router might influence the operation of another virtual router sharing the same CPU. The setup has two virtual routers, each one with a dedicated pair of physical interfaces and unidirectional traffic flow.

As a first step, we offer a load of 300 kpps on the ingress physical interfaces of both virtual routers at the same. We observe an aggregate throughput of 600 kpps. It shows that both virtual routers are working independently without affecting each other. The result is the same for both macvlan and NAPI-macvlan virtual routers. Thereafter, we increase the offered load to 1000 kpps on the ingress physical interface of VR1 while still offering 300 kpps on the ingress interface of VR2. For this scenario, the throughput results are presented in **Table 1**.

Table 1. Isolation between virtual routers.

Setup	Packet rate (kpps)					
	Offered load			Throughput		
	VR1	VR2	Total	VR1	VR2	Total
Macvlan	1000	300	1300	480	55	535
NAPI-Macvlan	1000	300	1300	350	300	650

We can note that the overall throughput for NAPI-macvlan is higher than for macvlan virtualization (650 kpps vs 535 kpps). Moreover, the overload on VR1 in the macvlan case results in serious performance degradation in VR2. In the NAPI-macvlan, on the other hand, no such effects can be seen.

The explanation to this difference in isolation can again be described by the backlog queue that is present in macvlan virtualization. Since the backlog queue is on a per CPU basis, it is shared between the two virtual machines. So, even though we have isolation at the physical interface level, this isolation cannot be preserved between VR1 and VR2 because of the shared backlog queue. For NAPI-macvlan, VR1 and VR2 do not have any drop location in common. Therefore, packet drops occur only on the physical interfaces and the isolation properties can be preserved.

4.5 Latency

Latency is an important parameter for many network applications. Pktgen provides a utility to compute packet latency. It records packet transmission time at the traffic generator and then packet reception time at the sink. The difference provides the packet latency. We use the case I setup for latency measurements. The test is conducted for a fixed amount of time (120 sec). A load of 600 kpps is offered and we make sure that all packets are received at the sink (i.e., no packet drop occurs here). **Fig. 9** displays the average latency for each configuration. It can be seen that the IP forwarder and NAPI-macvlan virtualization have the same latency. However, the average latency is doubled for macvlan virtualization.

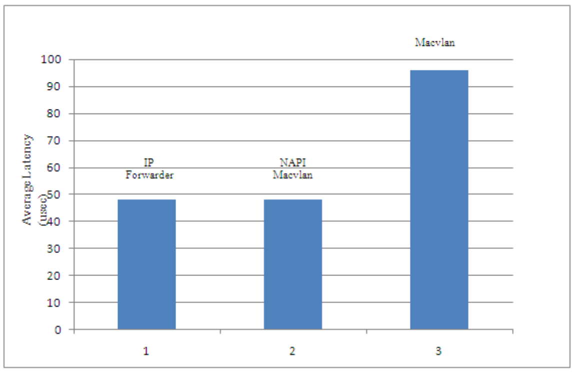

Fig. 9. Latency measurements for virtual setups

The reason for the higher average delay for the macvlan setup is that the macvlan uses the earlier mentioned Softnet API, incurring an overall processing delay. In case of both original IP forwarding and NAPI-macvlan, a packet is forwarded within a single SoftIRQ. This reduces the packet processing delay and results in a lower average latency.

5 Conclusion and Future Work

We have evaluated performance of virtual router platforms based on Linux Namespaces. The virtual routers were using the macvlan device (virtual interfaces) which is the part of the main stream Linux kernel 2.6. We analyzed the router's data plane and pointed out that backlog queuing can form a severe performance bottleneck. We proposed an alternative data plane by eliminating the backlog queue. To achieve this, we modified the macvlan device and introduced a variant denoted "NAPI-macvlan". We compared the performance of macvlan and NAPI-macvlan based virtual routers. We achieved better forwarding rates using NAPI-macvlan, particularly in different kinds of overload situations. Furthermore, in contrast to macvlan, NAPI-macvlan based routers proved superior when it comes to preserving isolation properties. It was also demonstrated that NAPI-macvlan based virtual routers improve the scaling properties. Finally, a considerable improvement in latency was also observed while using NAPI-macvlan. In our future work, we plan to evaluate NAPI-macvlan based routers using multiple CPU cores and multi-queue NICs.

Acknowledgments. We would like to thank Robert Olsson from TSLab KTH and Jens Laas from Uppsala University, for their valuable input during the work.

6 References

1. N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, L. Mathy, and T. Schooley, "Evaluating Xen for virtual routers," in PMECT07, August 2007.
2. F. Anhalt, P. Primet, "Analysis and experimental evaluation of data plane virtualization with Xen" IEEE 5th ICNS, November 2009.
3. N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, L. Mathy, "Towards High Performance virtual routers on commodity hardware", ACM CoNext December 2008.
4. Bavier, N. Feamster,M. Huang, L. Patterson and J. Rexford. "In VINI Veritas: Realistic and Controlled Network Experimentation", In SIGCOMM'06: Proceedings of ACM SIGCOMM 2006 Conference, September 11-15, 2006, Pisa, Italy.
5. S. Bhatia, M. Motiwala, W. Muhlbauer, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford, "Trellis: A Platform for Building Flexible, Fast Virtual Networks on Commodity Hardware" ACM ROADS'08, December 9,2008, Madrid, Spain.
6. E. Keller and E. Green. Virtualizing the data plane through source code merging. In PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow, 2008.
7. Y. Liao, D. Yin, L.Gao, "PdP: Parallelizing Data Plane in Virtual Network Substrate" ACM VISA'09, August 17,2009, Barcelona, Spain.
8. Y. Liao, D. Yin, L.Gao, "Crossbow: From Hardware Virtualized NICs to Virtualized Networks" ACM VISA'09, August 17,2009, Barcelona, Spain.
9. S. Rathore, M. Hidell, P. Sjödin, "Performance Evaluation of Open Virtual Routers" IEEE GlobeCom workshop on future Internet , December 10 2010, Miami, USA.
10. A. Bianco, J. M. Finochietto, G. Galante, M. Mellia and F. Neri, "Open-Source PC-Based Software Routers: A viable Approach to High-Performance Packet Switching", in LNCS 3375, pp. 353-366, Springer-Verlag Berlin Heidelberg 2005.
11. "Intel: Interrupt moderation using Intel Gigabit Ethernet controllers (Application Note 450)." Available http://download.intel.com/design/network/applnots/ap450.pdf , last accessed April, 2010
12. J.H.Salim, R.Olsson, A. Kuznetsov, " Beyond softnet", In: Proceedings of the 5th Annual Linux Showcase & Conference (ALS 2001), Oakland, CA, USA (2001).
13. M. Rio et al., "A map of the networking code in Linux kernel 2.4.20", Technical Report DataTAG-2004-1, FP5/IST DataTAG Project, Mar. 2004.
14. Ben Pfaff, Justin Petit, Teemu Koponen, Keith Amidon, Martin Casado, Scott Shenker "Extending Networking into the virtualization layer," ACM Sigcomm HotNets, September 2009.
15. S. Soltesz, H. Poltz, M. Fiuczynski, A. Bavier and L. Patersson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors", in EuroSys'07: Proceedings of the 2nd ACM EuroSys Conference, March (21-23) 2007.
16. RFC 2544 "Benchmarking methodology for interconnecting devices", http://tools.ietf.org/html/rfc2544, last accessed April, 2010
17. R. Olsson "pktgen the Linux packet Generator", Proceedings of the Linux Symposium Vol.2, pp. 11-24, July 20-23 2005, Ottawa.