American Journal of Networks and Communications
2013; 2(4): 88-96
Published online August 20, 2013 (http://www.sciencepublishinggroup.com/j/ajnc)
doi: 10.11648/j.ajnc.20130204.11

KVM vs. LXC: comparing performance and isolation of hardware-assisted virtual routers

Muhammad Siraj Rathore, Markus Hidell, Peter Sjödin

Network Systems Laboratory, School of ICT, KTH Royal Institute of Technology, Stockholm, Sweden

Email address:
siraj@kth.se (M. S. Rathore), mahidell@kth.se (M. Hidell), psj@kth.se (P. Sjödin)

To cite this article:
Muhammad Siraj Rathore, Markus Hidell, Peter Sjödin. KVM vs. LXC: Comparing Performance and Isolation of Hardware-assisted Virtual Routers. *American Journal of Networks and Communications* Vol. 2, No. 4, 2013, pp. 88-96. doi: 10.11648/j.ajnc.20130204.11

Abstract: Concerns have been raised about the performance of PC-based virtual routers as they do packet processing in software. Furthermore, it becomes challenging to maintain isolation among virtual routers due to resource contention in a shared environment. Hardware vendors recognize this issue and PC hardware with virtualization support (SR-IOV and Intel-VTd) has been introduced in recent years. In this paper, we investigate how such hardware features can be integrated with two different virtualization technologies (LXC and KVM) to enhance performance and isolation of virtual routers on shared environments. We compare LXC and KVM and our results indicate that KVM in combination with hardware support can provide better trade-offs between performance and isolation. We notice that KVM has slightly lower throughput, but has superior isolation properties by providing more explicit control of CPU resources. We demonstrate that KVM allows defining a CPU share for a virtual router, something that is difficult to achieve in LXC, where packet forwarding is done in a kernel shared by all virtual routers.

Keywords: Network Virtualization, Virtual Router (VR), SR-IOV, Virtual Function (VF), SoftIRQ, NAPI

1. Introduction

Network virtualization allows running heterogeneous virtual networks in parallel to support a diverse range of services over a shared substrate. An important building block of network virtualization is router virtualization. One way to enable virtual routers is to use open source virtualization technologies on commodity PC hardware and let each virtual machine act as a router. This is a flexible and low-cost solution. However, there are concerns that PC-based virtual routers could potentially suffer from low performance as packets are processed in software [1][2]. Furthermore, it becomes challenging to maintain isolation among virtual routers due to resource contention in a PC environment [3][4].

It is complicated to provide performance and isolation at the same time in a PC environment. To address this issue, we investigate how virtualization support in PC hardware can be used for virtual routers. Single root I/O virtualization (SR-IOV) [5] is a step in that direction. It provides hardware support to virtualize network interface cards (NICs). SR-IOV divides a single physical PCIe device into multiple PCIe instances, called Virtual Functions (VFs) [5][15]. A VF interface is an Ethernet-like interface that can be used in a virtual router. In addition, SR-IOV offloads packet handling from the host CPU and allows packets to be directly dispatched to virtual routers. This should result in performance improvements. Furthermore, SR-IOV provides dedicated hardware queues (receive and transmit) for each VF, which can be used to isolate traffic streams for different virtual routers.

Different virtualization approaches may have different properties from performance and isolation perspectives. KVM [6] is a full virtualization solution where hardware resources are virtualized through hypervisor software. It is a flexible solution that allows a diversity of virtual machines to run on the same host, but at the potential cost of performance penalties due to overhead. In contrast, LXC [7] is a container-based approach where operating system resources (e.g. files, system libraries, routing tables) are virtualized to create multiple execution environments within the same operating system [2][7]. It is attractive from a performance point of view, but has negative implications for isolation.

In our work we investigate the impact of SR-IOV on performance and isolation of KVM and LXC-based virtual routers. We anticipate a certain degree of performance improvement in both cases due to processing offload.

However, we believe that different virtualization techniques can exploit hardware support differently depending on how packets are processed along the forwarding path. Hence, we examine the forwarding path and modifications made by the hardware. Our hypothesis is that KVM could make use of hardware support more effectively by eliminating a considerable amount of software-based packet processing. In comparison, LXC is already lightweight in nature, and the relative improvement gains could be less significant.

When it comes to isolation, the resource contention among virtual routers may lead to poor isolation properties. For instance, an overloaded virtual router can consume much CPU and starve others. Such behavior can be avoided by defining a guaranteed CPU share for a virtual router. In this regard, Linux kernel introduces a CPU share feature to provide controlled CPU access among different processes [8]. However, CPU share is intended to control CPU for user-space applications (processes) in server environments, and may not be directly applicable for virtual routers. Generally, a forwarding path is a combination of many different components including interrupt processing, kernel and user level devices. It is not trivial to control CPU usage along a forwarding path in a consolidated fashion. In such situations, SR-IOV might be useful. It offloads packet handling by replacing software-based processing modules along the forwarding path. We investigate how CPU share can be combined with hardware-assisted forwarding paths in KVM and LXC-based virtual routers. Furthermore, we analyze how CPU guarantees can be enforced and what the effects would be on isolation between virtual routers. Finally, we identify an approach that provides a suitable trade-off between performance and isolation after hardware support.

The rest of this paper is organized as follows: Section 2 surveys related work on virtual router platforms. Section 3 presents the packet forwarding architecture for KVM and LXC-based virtual routers. Thereafter, section 4 describes our performance and isolation measurements, results and discussion. Finally, Section 5 concludes the paper.

2. Related Work and Contributions

There are several studies in the literature where Xen is proposed to enable virtual routers [9][10][11]. The work presented in [10] compares performance of two different versions of Xen (3.1 and 3.2). It shows that guest domain packet forwarding is improved in version 3.2, but that isolation is weakened when more virtual routers are added. Another work [11] suggests an architecture using Xen and Click. It achieves encouraging performance and isolation results, but requires dedicated NICs to be allocated to each virtual router, something that might be difficult to realize in practice. Others introduce customized components to improve forwarding performance [12], but observe degraded isolation (in terms of packet loss) at high network load.

We conclude from previous work that software-based solutions have difficulties providing high performance and strong isolation at the same time. We therefore propose a hardware-assisted platform for router virtualization. There are some examples where SR-IOV is proposed to improve performance of virtual machines [13][14] in a server setting. The focus of these studies is not on virtual routers and therefore no relevant results are available (e.g. packet forwarding rate, latency etc).

In our previous work, we study PC-based virtual routers [2][15]. The first study [2] compares two container-based approaches (i.e. OpenVZ and LXC) from a performance perspective. Our results show that LXC achieves better performance than OpenVZ. However, the work does not consider hardware assistance for virtualization. The other work [15] focuses on hardware assistance (i.e. SR-IOV) to enable virtual routers in LXC environment. We investigate how SR-IOV can be used to enable parallel forwarding paths over a multi-core platform. We dedicate a CPU core to each virtual router in order to improve performance and isolation. The focus of the current paper is different, since it aims to compare two different virtualization techniques. In addition, we evaluate a more challenging and practical scenario where a CPU core is shared between virtual routers. We anticipate that such resource sharing may result in degraded isolation properties for a shared kernel environment (i.e. LXC). In this regard, a KVM-based approach might be a better alternative, since it uses a different packet forwarding architecture.

To our knowledge, there is no previous work on comparing KVM and LXC for hardware-assisted virtual routers. These techniques are part of the mainstream Linux kernel and hence readily available. A solution based on them should be more up to date and adoptable. On the other hand, Xen based solutions in existing literature are not completely in line with main stream kernel. For instance, Xen uses custom process scheduler and memory management system.

We evaluate the CPU share feature of Linux kernel to control CPU usage of parallel running virtual routers, something not investigated before. In addition, we study how non-virtualized components of the underlying host (e.g. NAPI will be discussed later) can impact the behavior of virtual routers.

3. Virtual Routers Forwarding Architecture

In this section we explore the packet forwarding path for KVM and LXC-based virtual routers. First we discuss various software-based devices to enable virtual routers. Then we investigate how these devices can be replaced with hardware-assisted devices. We analyze impact of hardware support on forwarding paths from performance and isolation perspectives.

3.1. KVM-based Virtual Routers

Figure 1 shows the packet forwarding path for a KVM-based virtual router. When a packet is received on a physical network interface (NIC1) it is copied to the main memory of the host system, and an interrupt is generated to notify the CPU. The Linux packet reception API (NAPI [16]) handles this interrupt. It adds the network interface to a queue (the NAPI poll list for interfaces with incoming traffic) and disables interrupts for more incoming packets on that interface. An RX SoftIRQ (software interrupt/kernel thread) is scheduled to process the packet. Through the SoftIRQ, NAPI serves network interfaces (i.e. packet processing) from the NAPI poll list in a round robin fashion.

In a virtualized environment, the first processing task is to identify the virtual interface (VIF) within the virtual router that should process the incoming packet. A VIF is an Ethernet-like interface with a unique MAC address. A virtual router may contain one or more VIFs. The VIF is identified based on the destination MAC address in the Ethernet header. This process is known as physical-virtual *device mapping* (Figure 1).

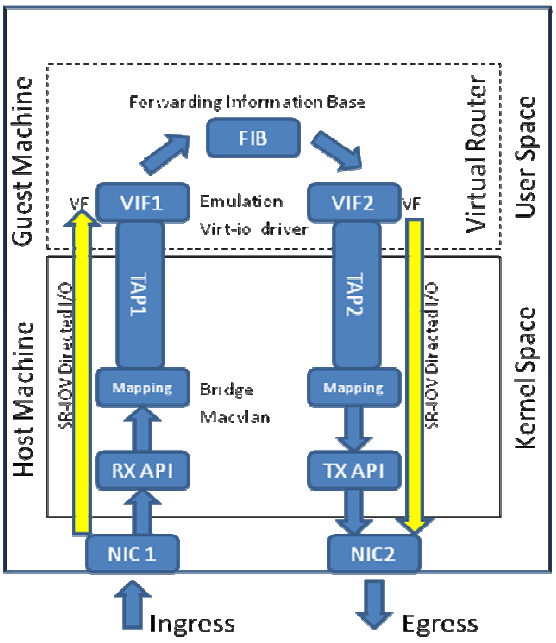

Figure 1. *Forwarding path- KVM-based virtual router*

There are various options for how to perform device mapping. The most common is Linux bridging. It is a software implementation of a bridge function that can switch packets between any pair of interfaces based on MAC addresses. However, Linux bridging has a considerable amount of overhead for redirecting packets between virtual and physical interfaces. Similar solutions are virtual switch [17] and Open vSwitch [18]. An attractive solution from a performance perspective is to replace the software bridge with macvlan devices. A macvlan device maintains a static MAC address table to provide physical to virtual address mapping and thus incurs less processing overhead [2].

After the device mapping, the packet should be made available to the virtual router on the VIF. However, according to the KVM architecture, a guest machine runs in user space with its own memory management system [6]. This requires a copy operation to move the packet from kernel to the user memory that belongs to the VIF. Furthermore, a CPU context switch is also required from kernel to user mode. Hence, a packet cannot be immediately delivered to the VIF after device mapping. Instead, it must be queued in kernel space. We use the tap device for this purpose (Figure 1). The tap device is a layer 2 network device that consists of two interfaces, one in kernel space and one in user space. The two interfaces are connected in such a way that data written at one end is available for reading at the other end.

At this point there are several alternatives for a VIF. For instance, a virtual interface can be an emulated network device using Qemu. An emulated device can be attractive as it uses standard network device drivers without any changes (e.g. Intel e1000). However, emulation is a CPU-intensive task and leads towards high performance penalties. Another option is the virtio para-virtualized network device. It is optimized to reduce virtualization overhead but requires modifications in the guest kernel. Yet another option is the macvtap device. It integrates the functionality of macvlan (device mapping), tap device and VIF in a single device, which makes it an attractive choice.

Once the packet is available on the ingress VIF (VIF1 in Figure 1) of the virtual router, the forwarding decision is taken and next hop is determined. The packet is placed on the outgoing virtual interface VIF2. After that, the tap interface and the mapping device are used in order to place the packet on the outgoing physical interface (NIC2).

We conclude that the software-based approach demands much packet processing in order to deliver a packet to a virtual router, which could lead to performance penalties. In contrast, a hardware-assisted approach makes it possible to directly deliver a packet to a virtual interface inside a virtual router without any major intervention from the system CPU. With such an approach, when a packet is received on a NIC, it is passed to a hardware switch inside the NIC. A destination MAC address lookup is performed to determine the VF. After that, the packet is placed on a hardware queue reserved for that VF. The next step is to transfer the packet from NIC to the VF's memory areas in the virtual router (i.e. the guest machine). The guest machine has its own memory addresses separated from host memory addresses, so guest memory addresses need to be translated to host memory addresses in order to perform a DMA operation. Hardware support for such address translation is integrated inside the CPU chipset [19] (Intel-VTd, directed I/O), which makes it possible to directly transfer a packet to virtual router memory (something that would otherwise require software intervention). This should improve forwarding

performance.

We see in Figure 1 that SR-IOV replaces many software-based devices (e.g. bridge, macvlan, and tap) and therefore appears promising for considerable processing offload. We also observe other architectural benefits; SR-IOV replaces the entire kernel level packet processing required in the software-based solution. Packets are processed only in user space. This should improve cache locality and reduce the amount of CPU context switches (between kernel and user space) compared to a software-based architecture. In addition, it should be more straight-forward to control a CPU in such an architecture, by for example defining CPU share for user space processes. As a result, we expect to achieve better isolation between virtual routers.

3.2. LXC-based Virtual Routers

In the case of LXC, packet reception and device mapping are similar to that of KVM. However, in contrast to KVM, LXC performs packet forwarding in kernel space, as shown in Figure 2. It does not require any packet copying or context switching operations, and the packet is immediately available to a virtual router after device mapping. This should result in better performance, compared to KVM.

We have two choices for VIFs. The first is the virtual Ethernet (veth) device. It is an Ethernet-like device, which can be used in combination with the bridge device in order to access the host's physical devices. Alternatively we can use the macvlan device, which combines the functionality of virtual interfaces and device mapping.

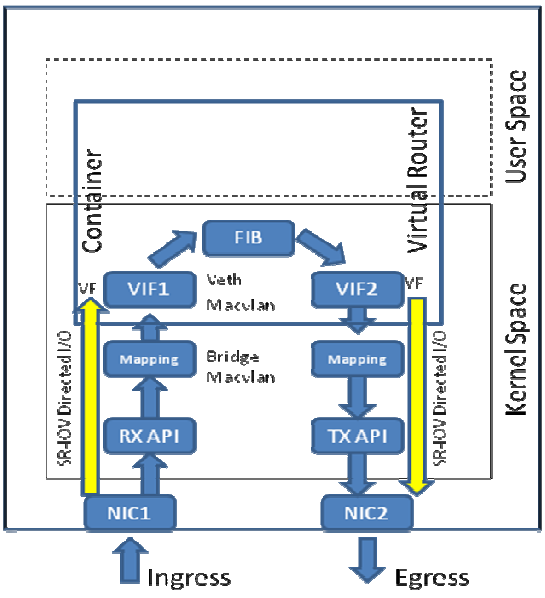

Figure 2. *Forwarding path- LXC-based virtual router*

We see in Figure 2 that SR-IOV can be used in the same way as for KVM to offload packet processing onto the hardware. However, in contrast to KVM, there are not many differences between software and hardware based solutions from an architectural point of view, since both solutions perform packet processing in kernel space. The kernel space processing might be an advantage from a performance point of view, but there are potential drawbacks from a resource management perspective. The reason is that packet handling inside the kernel is mainly done through SoftIRQ processing. SoftIRQ serves all (virtual) interfaces in a round robin fashion with equal priority, even though the interfaces may belong to different virtual routers. Accordingly, it is not possible to control the CPU usage of individual forwarding paths, something that we expect could lead to poor isolation properties.

4. Experimental Evaluation

In this section we evaluate performance and isolation of KVM and LXC-based virtual routers. First, we measure performance both for software and hardware based solutions. For KVM, we compare macvtap and SR-IOV whereas macvlan and SR-IOV are compared in LXC. We investigate the level of performance gains that can be achieved using hardware assistance. As the next step we increase the number of virtual router running in parallel and measure the effect on aggregated performance.

For the isolation study, we consider two SR-IOV-based virtual routers running in parallel with different performance requirements. We overload one of the virtual routers and study the impact on the performance of the other virtual router. We evaluate two different scenarios by varying performance requirements and offered loads.

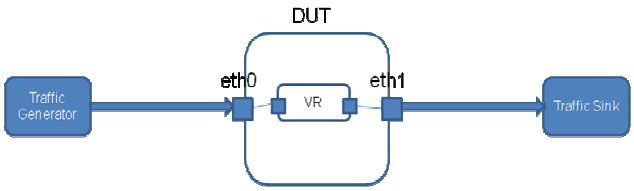

Figure 3. *Experimental test bed*

The experimental setup is shown in Figure 3. We use three Linux machines where the first, Traffic Generator, is used to generate network load using pktgen [20]. The load is fed into the device under test (DUT). The DUT has two physical interfaces and a virtual router that forward packets from one interface to the other. The virtual router is configured with two virtual interfaces as shown in Figure 3; one virtual interface is connected to the physical ingress interface while the other virtual interface is connected to the physical egress interface. The network load is received on a third machine, Traffic Sink. All performance measurements are taken at Traffic Sink using pktgen receiver side utility [21].

The hardware used for DUT is Intel i7 Quad Core 3.4 GHz processor (Intel VT-d supported, chipset Intel Q-67 Express) and 4GB of RAM, running Linux kernel net-next 3.2-rc1. We use a single CPU core in all experiments unless otherwise stated. The machine is also equipped with one 1 Gbps dual-port NIC with an Intel 82576 GbE controller. On

each port a maximum of eight SR-IOV devices are supported. It means that we can run up to eight parallel virtual routers using the configuration shown in Figure 3.

4.1. Performance Results

We offer network load (100 UDP Flows) on DUT and gradually increase the load until line rate i.e. 1488 kilo packets per second (kpps) using 64 byte packets. As a baseline, we relate the performance of the virtual router to the performance of regular "IP forwarding" in a non-virtualized Linux-based router.

The throughput results are presented in Figure 4. It can be seen that only the baseline IP forwarder is able to achieve line rate. The rest of the configurations are below line rate. SR-IOV has the highest rate compared to the other software-based approaches. It is interesting to note that the difference between SR-IOV (KVM) and SR-IOV (LXC) is marginal. The former achieves 1250 kpps whereas SR-IOV (LXC) obtains 1300 kpps. It indicates that full virtualization with proper hardware support can achieve performance comparable to that of lightweight containers.

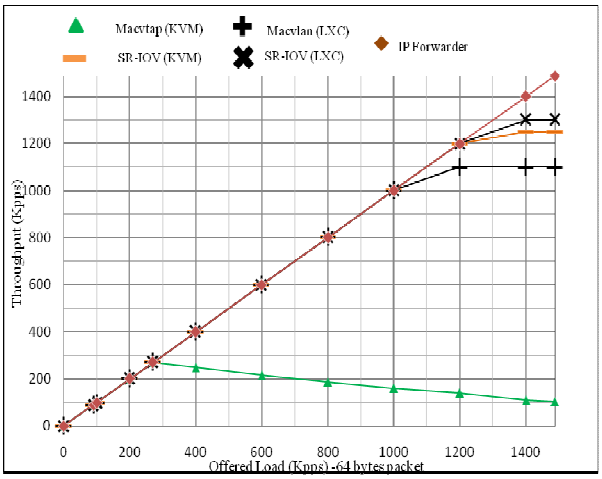

Figure 4. *Throughput- LXC vs. KVM based virtual router*

A clear performance difference can be observed among the software-based approaches. The macvlan (LXC) achieves around 11 times higher throughput than macvtap (KVM). This is remarkable, so we investigate more closely the poor performance of macvtap. With macvtap, there is a significant amount of packet drop on the tap interface (kernel side) at high packet rates. The amount of packet drop increases when the load is increased. This results in more throughput degradation. For instance, we see in Figure 4 that throughput is around 280 kpps for macvtap (KVM) at an offered load of 280 kpps. However, when load is increased up to the line rate, throughput degrades to 100 kpps.

We further investigate the reasons behind this large packet drop by measuring how CPU utilization varies with offered load. In addition to total CPU utilization, we also measure CPU consumption in kernel and user space. We see in Figure 5 that macvtap-kernel CPU usage increases very quickly with offered load. This behavior points towards an architectural bottleneck of the KVM software-based setup: In the software-based approach, packets are switched between kernel and user space during forwarding and this switching becomes a bottleneck at high load. The packet handling is done through SoftIRQs inside the kernel, which runs at higher CPU priority than user space processes. At high offered load, the CPU is occupied with SoftIRQ processing most of the time. This results in starvation of user space processes and the virtual router is unable to process its incoming queue. As a result, packets are simply dropped after RX SoftIRQ handling, without further processing. This is clearly a waste of CPU resources and results in throughput degradation. The SR-IOV (KVM) eliminates this bottleneck by offloading the kernel side packet handling to hardware. This results in much higher throughput.

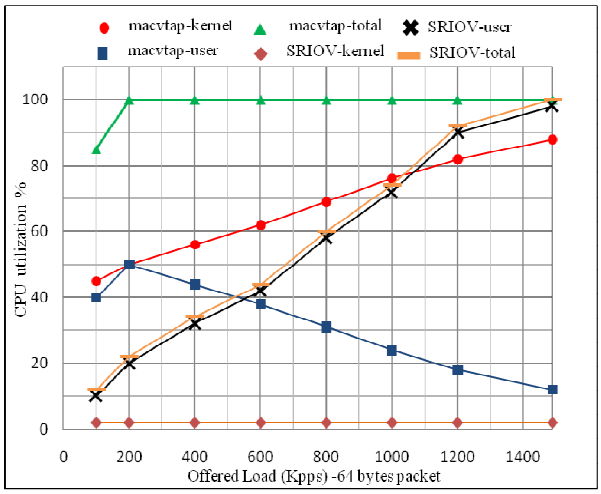

Figure 5. *CPU utilization of KVM based virtual router*

In addition to throughput, we measure latency at maximum offered load (i.e. line rate). The results follow the same pattern as for the throughput measurements. We see in Figure 6 that SR-IOV is very effective for KVM and produces comparable results to SR-IOV (LXC).

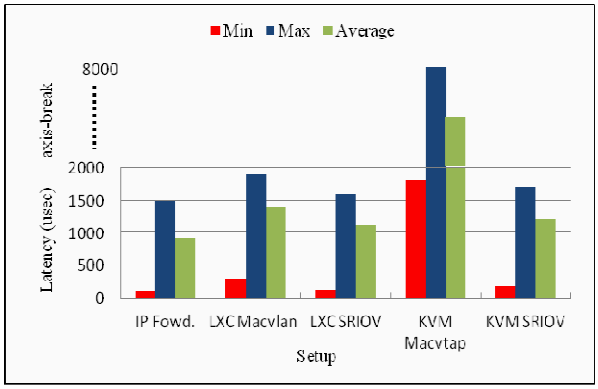

Figure 6. *Latency- LXC vs. KVM based virtual router*

As a next step, we gradually increase the number of virtual routers running in parallel. We offer load at line rate

and measure the aggregated throughput. We see in Figure 7 that throughput for the macvtap case drops to zero with five virtual routers. It shows complete starvation of user space tap devices as a result of extensive SoftIRQ processing. The performance is quite reasonable for the rest of the configurations. It appears that SR-IOV (LXC) scales better than SR-IOV (KVM). The performance difference becomes more pronounced as the number of virtual routers increases. The SR-IOV (LXC) achieves 1230 kpps whereas SR-IOV (KVM) obtains 949 kpps for eight parallel virtual routers. The difference is considerable; still, SR-IOV (KVM) is achieving reasonable performance at high-load conditions (i.e. 64-byte packets).

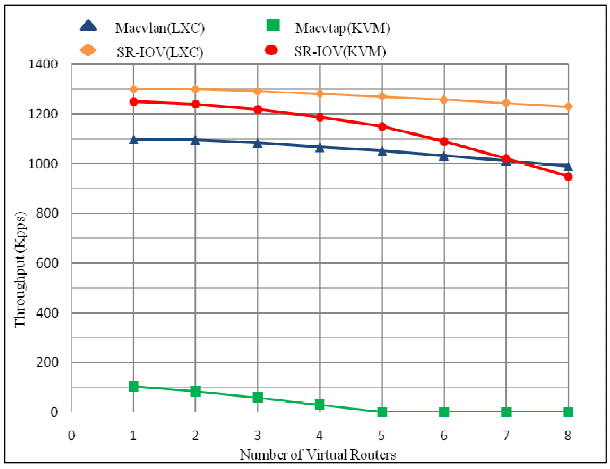

Figure 7. *Throughput vs. no. of virtual routers (1 CPU core)*

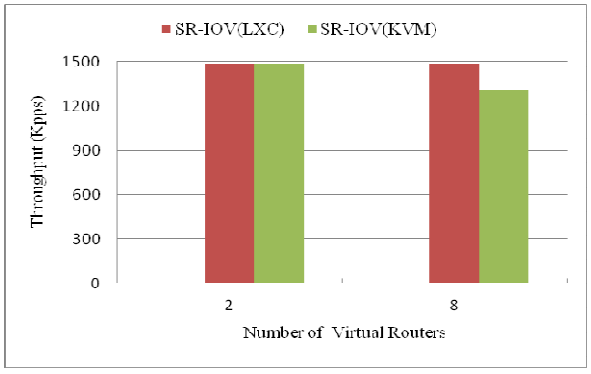

Figure 8. *Throughput vs. no. of virtual routers (2 CPU cores)*

As the next step, we investigate how performance scales while introducing another CPU core. We configure a CPU core to process all traffic belonging to one virtual router while the other core services a second virtual router. It can be seen in Figure 8 that line rate (1488 kpps) is achieved both for SR-IOV (LXC) and SR-IOV (KVM). However, some performance drop can be seen while adding more virtual routers for SR-IOV (KVM). Apart from this, performance is still quite high and we observe performance scalability for both cases. It can also be noticed that we are approaching line rate and adding more CPU cores probably would not increase throughput much. In this regard, it might be interesting to test performance scalability over a 10Gbps network.

4.2. Isolation Results

For the isolation experiments we focus on SR-IOV (KVM) and SR-IOV (LXC). We consider two identical virtual routers VR1 and VR2, with two VFs on each virtual router (Figure 9). We offer network load on eth0. The two virtual routers are responsible for processing packets in parallel and for forwarding them onto the same outgoing interface eth1.

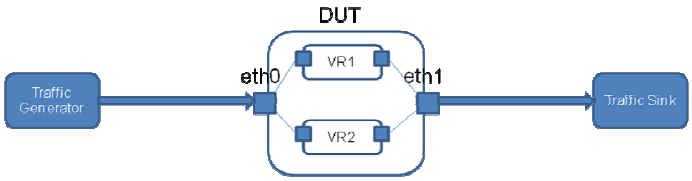

Figure 9. *Isolation test setup*

In this setup we use a single CPU core for both virtual routers, in order to explore possible CPU contention under stress and then investigate different ways of resolving it. We consider two different scenarios of router overload:

4.2.1. Scenario I: VR1 overloaded, VR2 fixed at 400 kpps

In this scenario we assume that the performance requirement for VR2 is 400 kpps. The VR1 is free to attain as much throughput as possible. However, the objective is that the load conditions on VR1 should not affect the performance of VR2. We offer a network load of 400 kpps towards each virtual router. The aggregated offered load on the DUT is 800 kpps. At this point, we gradually overload VR1 and study the impact on the performance of VR2. The offered load for VR1 is increased up to 1088 kpps (when a line rate of 1488 kpps on eth0 is reached) while it remains at 400 kpps for VR2.

The results are shown in Table 1 and Table 2 for LXC and KVM respectively. We can see that isolation is maintained both for LXC and KVM. The performance for VR2 remains at 400 kpps no matter the load conditions on VR1.

We believe that the high degree of isolation of LXC comes from the NAPI RX API in the host kernel. In our setup, we have incoming traffic on two VFs belonging to two different virtual routers. This means that the host kernel is responsible for handling incoming traffic on two interfaces in parallel. The NAPI algorithm maintains fairness among network interfaces that share a CPU [16]. The network interfaces are served in a round robin fashion during a RX SoftIRQ. The NAPI processes only a certain number of packets for an interface before it switches to serve the other interface. In this way it is not possible for an interface to monopolize the CPU. As a result, in an overload situation the excessive packets are simply dropped. This may result in some throughput degradation but provides isolation between interfaces.

The strong isolation properties of KVM may come from

Linux completely fair process scheduler (CFS) [8]. KVM-based virtual routers (which are simple user space processes as discussed in Section 3) are scheduled using CFS. The CFS maintains a mechanism (i.e. red block tree) to impose certain degree of fairness in CPU time allocation among processes. A process that receives less CPU time is given priority over those that have consumed more CPU time. As a result all running processes receive a fair amount of CPU time.

Table 1. *LXC: VR1 overloaded and VR2 fixed at 400 kpps*

Offered load (kpps)			**CPU%**	**Throughput (kpps)**		
VR1	VR2	Total	Total	VR1	VR2	Total
400	400	800	61	400	400	800
600	400	1000	75	600	400	1000
800	400	1200	85	705	400	1105
1000	400	1400	100	715	400	1115
1088	400	1488	100	720	400	1120

Table 2.*KVM: VR1 overloaded and VR2 fixed at 400 kpps*

Offered load (kpps)			**CPU%**			**Throughput (kpps)**		
VR1	VR2	Tot	VR1	VR2	Tot	VR1	VR2	Tot
400	400	800	33	33	66	400	400	800
600	400	1000	49	33	82	600	400	1000
800	400	1200	60	33	93	800	400	1200
1000	400	1400	67	33	100	836	400	1236
1088	400	1488	67	33	100	842	400	1242

4.2.2. Scenario II: VR2 overloaded, VR1 fixed at 800 kpps

We see in scenario I that isolation is achieved thanks to built-in fairness policies in Linux kernel. However, for other scenarios where for instance one virtual router should be given priority over other, fairness policies might be less suitable.

In order to test our hypothesis we make some changes to scenario I. We still assume a performance requirement of 400 kpps for VR2. In addition, we consider VR1 with a performance requirement of 800 kpps. However, here we overload VR2 instead. The offered load for VR2 is increased from 400 kpps to 688 kpps whereas a constant load of 800 kpps is offered on VR1. Ideally, the overload conditions on VR2 should not degrade VR1 performance.

The results are shown in Table 3 and Table 4 for LXC and KVM respectively. We notice that both LXC and KVM yield poor isolation. In LXC, we observe that VR1 performance decreases from 719 kpps to 667 kpps whereas it increases from 400 kpps to 633 kpps for VR2. In KVM, we notice that VR1 performance decreases from 800 kpps to 682 kpps whereas it increases from 400 kpps to 567 kpps for VR2. The poor isolation is related to the lack of sufficient CPU resource for VR1. For instance we see in Table 4 that 61% CPU is required to support 800 kpps. However, when the load increases on VR2 the CPU usage for VR1 drops to 53%, as a result of CPU contention between VR1 and VR2. As a result, we observe performance degradation for VR1.

In order to achieve the required isolation, we should give priority to VR1 over VR2 in terms of CPU time. However, this is hard in LXC for two reasons. First, it is difficult to control CPU usage for each virtual router in a shared kernel. This is the reason why we are unable to present such data in Table 1 and Table 3. Secondly, LXC uses NAPI for packet processing in host kernel. The fairness policy of NAPI algorithm provides equal CPU time to the virtual routers. There is no easy way to adapt NAPI's behavior to our requirements. In the first scenario, our performance requirements match with NAPI behavior and we achieve isolation. However, in the second scenario, it is not the case and we observe poor isolation. We also notice that the CPU share feature has no control over shared-kernel forwarding paths as it is intended for user space processes.

In contrast to LXC, the KVM virtual router is a user space process and hence priority can be given to any virtual router using CPU share. When CPU share is configured, it changes CFS from a "fair" to a "proportional weight" scheduler. We allocate a CPU share to VR1 that is two times to the CPU share of VR2. It allows VR1 to obtain at least 66.66% of CPU time whereas VR2 is allowed to get at least 33.33%. The behavior of CPU share is work-conservative, which means that a VR is privileged to consume any unused share (or a portion of share) of the other VR(s). We repeat our experiment with these settings and the results are shown in Table 5. We see that both virtual routers achieve the required performance even in overload situations. VR1 is ensured its required CPU share regardless the load conditions on VR2. We also notice that VR1 consumes less than its allocated share and that an additional share is used by VR2 without causing any isolation problems.

Table 3. *LXC: VR2 overloaded and VR1 fixed at 800 kpps*

Offered load (kpps)			**CPU%**	**Throughput (kpps)**		
VR1	VR2	Total	Total	VR1	VR2	Total
800	400	1200	87	719	400	1119
800	500	1300	100	708	500	1208
800	600	1400	100	687	600	1287
800	688	1488	100	667	633	1300

Table 4. *KVM: VR2 overloaded and VR1 fixed at 800 kpps*

Offered load (kpps)			CPU%			Throughput (kpps)		
VR1	VR2	Tot	VR1	VR2	Tot	VR1	VR2	Tot
800	400	1200	61	33	94	800	400	1200
800	500	1300	59	41	100	778	460	1238
800	600	1400	56	44	100	739	502	1241
800	688	1488	53	47	100	682	567	1249

Table 5. *KVM CPU share: VR2 overloaded and VR1 fixed at 800*

Offered load (kpps)			CPU%			Throughput (kpps)		
VR1	VR2	Tot	VR1	VR2	Tot	VR1	VR2	Tot
800	400	1200	60	33	93	800	400	1200
800	500	1300	61	39	100	800	441	1241
800	600	1400	60	40	100	800	443	1243
800	688	1488	61	39	100	800	440	1240

5. Conclusions and Future Work

In this paper we compare KVM and LXC as means to enable virtual routers. We investigate the level of performance that can be gained using hardware support for virtualization. The results show that hardware support is especially effective for full virtualization (KVM). We find that switching packets between kernel and user space is a potential bottleneck for the software-based KVM approach at high offered loads. The hardware assistance makes it possible to perform the entire packet processing in user space. This alleviates the bottleneck and we can see a significant performance improvement (Figure 4). In comparison, the behavior of LXC is somewhat different. We see some performance gain when moving from a software-based to a hardware-assisted approach. The gain is expected as we offload some packet processing (e.g. device mapping) onto hardware. However, in contrast to KVM, the hardware assistance does not constitute any major architectural changes, since packet processing is still done in the shared kernel. The shared kernel makes it hard to restrict the CPU allocation for a particular virtual router, which leads to very limited possibilities for isolating virtual routers from each other. In contrast, the KVM hardware-assisted architecture achieves a much higher degree of isolation between virtual routers. It is sufficient to control the CPU share for a virtual router process running in user space.

For future work, we plan to enable multiple virtual networks with QoS guarantees for different types of services on a shared substrate.

References

[1] J. Whiteaker, F. Schneider, R. Teixeira, "Explaining packet delays under virtualization," ACM SIGCOMM Computer Communication Review, Vol. 41 Number 1, January 2011.

[2] S. Rathore, M. Hidell, P. Sjödin, "Performance Evaluation of Open Virtual Routers," IEEE GlobeCom workshop on future Internet, Miami USA, December 2010.

[3] S. Rathore, M. Hidell, P. Sjödin, "Data Plane Optimization in OpenVirtual Routers," IFIP Networking, Valencia Spain, May 2011.

[4] G. Somani, S. Chaudhary, "Application performance isolation in virtualization," IEEE International Conference on Cloud Computing, Bangalore India, September 2009.

[5] PCI-SIG: PCI-SIG Single Root I/O Virtualization Specifications, http://www.pcisig.com/specifications/iov/single_root/

[6] A.Kivity, Y.Kamay, D.Laor, "KVM: Linux virtual machine monitor," Proceedings of Linux Symposium, Ottawa Canada, June 2007.

[7] Linux Namespaces, http://lxc.sourceforge.net/index.php/about/kernel-namespaces/

[8] Linux process scheduler, https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

[9] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, L. Mathy, and T. Schooley, "Evaluating Xen for virtual routers," IEEE ICCCN workshop on Performance Modeling and Evaluation in Computer and Telecommunication Networks (PMECT07), Honolulu USA, August 2007.

[10] F. Anhalt, P. Primet, "Analysis and experimental evaluation of data plane virtualization with Xen," IEEE International Conference on Networking and Services (ICNS), Valencia Spain, April 2009.

[11] Greenhalgh, M. Handley, L. Mathy, N. Egi, M. Hoerdt, and F. Huici, "Fairness issues in software virtual routers," ACM SIGCOMM PRESTO Workshop, Seattle USA, August 2008.

[12] S. Bhatia et al. "Hosting virtual networks on commodity hardware," Georgia Tech. University, Tech. Report, GT-CS-07-10, January 2008.

[13] J. Liu, "Evaluating Standard-Based Self-Virtualizing Devices: A Performance Study on 10 GbE NICs with SR-IOV support," IEEE International Symposium on parallel and distributed processing, Atlanta USA, 2010.

[14] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, H. Guan, "High Performance Network Virtualization with SR-IOV," IEEE International Symposium on High Performance Computer Architecture, Banglore India, 2010.

[15] S. Rathore, M. Hidell, P. Sjödin, "PC-based Router Virtualization with Hardware Support," IEEE International Conference on Advanced Information Networking and Applications (AINA), Fukuoka Japan, March 2012.

[16] J.H.Salim, R.Olsson, A.Kuznetsov, "Beyond softnet," Proceedings of the 5th Annual Linux Showcase &

Conference (ALS 2001), Oakland USA, 2001.

[17] Ben Pfaff, Justin Petit, Teemu Koponen, Keith Amidon, Martin Casado, Scott Shenker, "Extending Networking into the virtualization layer," ACM SIGCOMM HotNets Workshop, New York USA, September 2009.

[18] The Openvswitch Project, http://openvswitch.org/

[19] Intel Virtualization Technology, http://www.intel.com/technology/itj/2006/v10i3/2-io/7-conclusion.htm

[20] R. Olsson, "pktgen the Linux packet Generator," Proceedings of the Linux Symposium, Vol.2 pp. 11-24, Ottawa Canada, July 2005.

[21] D. Turull, "Open source traffic analyzer," Master's thesis, KTH Information and Communication Technology, 2010. http://tslab.ssvl.kth.se/pktgen/docs/DanielTurull-thesis.pdf