Shrinking the Hypervisor One Subsystem at a Time

A Userspace Packet Switch for Virtual Machines

Julian Stecklina
TU Dresden
jsteckli@os.inf.tu-dresden.de

Abstract

Efficient and secure networking between virtual machines is crucial in a time where a large share of the services on the Internet and in private datacenters run in virtual machines. To achieve this efficiency, virtualization solutions, such as Qemu/KVM, move towards a monolithic system architecture in which all performance critical functionality is implemented directly in the hypervisor in privileged mode. This is an attack surface in the hypervisor that can be used from compromised VMs to take over the virtual machine host and all VMs running on it.

We show that it is possible to implement an efficient network switch for virtual machines as an unprivileged userspace component running in the host system including the driver for the upstream network adapter. Our network switch relies on functionality already present in the KVM hypervisor and requires no changes to Linux, the host operating system, and the guest.

Our userspace implementation compares favorably to the existing in-kernel implementation with respect to throughput and latency. We reduced per-packet overhead by using a run-to-completion model and are able to outperform the unmodified system for VM-to-VM traffic by a large margin when packet rates are high.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.4.4 [*Operating Systems*]: Communications Management–Network communication

Keywords Virtualization; Networking; Security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VEE '14, March 1–2, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2764-0/14/03…$15.00.
http://dx.doi.org/10.1145/2576195.2576202

1. Introduction

A large share of services on the Internet and in private datacenters run in virtual machines [4]. To achieve efficiency, virtualization solutions, such as Qemu/KVM [3], have moved toward a monolithic system architecture in which performance critical functionality is implemented directly in the hypervisor.

Monolithic virtualization layers are a particularly attractive target for attacks, because programming errors can lead to denial of service, disclosure of confidential information, or complete takeover of the virtual machine host by a malicious third party including every unrelated virtual machine on the same host. Security of the virtualization layer is of the utmost importance.

The motivation for having a lean hypervisor with small attack surface can be drawn from the microkernel community, which argues to separate as many components as possible in their own address spaces and thus introduce additional security hurdles for attackers. Despite growing experience in designing microkernel-based systems [8] and applying these design principles to virtualization [28], commodity hypervisors have not adopted the microkernel approach.

In addition to mitigating safety and security concerns, running subsystems in userspace also offers an easy way to update, inspect, and debug them with tools that are familiar to developers of ordinary userspace programs. Yet we still observe the ongoing adherence to monolithic systems even for operating system kernels in general, despite successful efforts to move classic kernel functionality into userspace [13].

Networking is a particularly important and problematic [31] aspect of an efficient virtualization layer. In this paper, we show that an efficient userspace implementation can be achieved for the network path in a commodity hypervisor, and more specifically that network connectivity for virtual machines is possible without any network specific code at all running in the hypervisor or in privileged components.

Our example platform is KVM [14], an efficient and mature virtualization solution for Linux. KVM is interesting in this context, because it relies on the userspace vir-

tual machine monitor Qemu [3]. The per-VM Qemu process provides additional isolation between VMs, but the role of Qemu has shrunk since KVM's inception. As of Linux 3.10 everything regarding interrupt handling and timeouts [18], except their initial setup, is for the sake of performance now done by the KVM module itself. This includes instruction decoding and device emulation of certain timers and interrupt controllers.

With the introduction of `vhost-net`, an accelerator module for paravirtualized networking, network I/O handling is handled in the Linux kernel. Future Linux versions will have in-kernel implementations of paravirtual block I/O as well. Qemu is left to handle the VM bootstrap and any devices that are not deemed worthy of a kernel implementation yet, while the attack surface in the most privileged software component in the system, the hypervisor, is growing.

We introduce *sv3*, a network packet switch implemented as a normal Linux process, which is a replacement for the `vhost-net` acceleration module in the Linux kernel, an in-kernel packet switch, and the `virtio` implementation in Qemu itself. sv3 can also host the driver for the physical network adapter used for outgoing traffic in userspace.

sv3 concentrates networking functionality in a single process, but has all the advantages mentioned earlier. Isolated networking islands can be built by running multiple sv3 instances. Faults in one instance do not spread to subsystems, processes, or VMs that do not depend on it.

Our paper makes the following contributions:

- We design a userspace network switch for virtual machines that relies on existing functionality in Linux (Section 3).
- We evaluate our prototype switch (Section 4) and show that the performance of such a design is comparable to a production in-kernel implementation, while security properties are superior.
- We conclude that implementation of the networking path in the hypervisor itself, including (para-)virtual device emulation, packet switching, and driver for the upstream network adapter is not necessary, because userspace implementations can be equally efficient in a commodity system.

In the following section, we review how Qemu and KVM implement networking. The informed reader may skip to the design of sv3 in Section 3.

2. Networking in Qemu/KVM

Bellard [3] developed Qemu as a standalone emulator for all essential devices of a complete PC[1] and as thus needed no special support by the kernel initially, but suffered high overhead. This situation was resolved by the advent of hardware-assisted virtualization on the x86 platform [29], which made it possible to use virtualization with close to native speeds.

KVM [14], a small abstraction for hardware virtualization features, was introduced to make use of hardware virtualization features and Qemu was modified to support it. The combination of Qemu and KVM is an efficient and stable production virtualization layer.

2.1 `virtio`

Given that the network is usually the only way for a virtual machine to interact with the outside world, access to the network has to be particularly efficient. `virtio` is a network interface specification especially suited for use in virtual machines. It works like other modern NICs by offering the guest a set of DMA queues, at least one receive (RX) queue and one transmit (TX) queue. The guest then chains multiple DMA descriptors together to form a descriptor chain, which represents either a buffer to receive packet data or a buffer that contains a packet ready to be sent. Whenever the guest needs to notify the host, either after queuing packets for transmissions or offering buffers for packet receiption, the guest writes to the `NOTIFY` register of the `virtio-net` device, which may be either an I/O port or a MMIO register depending on the hardware platform. Similarly, if the host needs to notify the guest, it injects IRQs.

The specification offers a way for both parties to see whether the other needs to be notified. Each queue has `NO_NOTIFY` and `NO_INTERRUPT` bits that can be set, when explicit notifications are not necessary.

If both guest and host support offloads, a `virtio-net` device offers a full complement of stateless offloads to the guest, such as checksum and TCP segmentation offload for sending and large receive offload for receiving packets.

A complete description of `virtio` is out of scope for this paper. We refer the reader to Russel [24].

2.2 Accelerating `virtio`

Even with KVM, until the introduction of `vhost-net`, Qemu handled network I/O in userspace and used `tap` devices to pass packets to the kernel. These `tap` devices are usually bridged with a physical NIC to provide access to a network. For a `virtio` network device, sending a packet involves exiting the virtual machine and scheduling the Qemu process. Qemu will pass the packet to the `tap` device with a `write` system call. When sending packets between VMs, this scheme results in up to four system calls and multiple packet copies.

`vhost-net` was introduced to Linux in 2010 in order to reduce virtualization overheads for network-heavy workloads. The main idea is to move the packet handling path of the `virtio` backend into the kernel and remove unnecessary mode switching and packet copying.

Instead of tying `vhost-net` to KVM and creating a special purpose solution, the Linux developers used mul-

[1] Of course, Qemu also supports different architectures, such as ARM, MIPS, PowerPC and others.

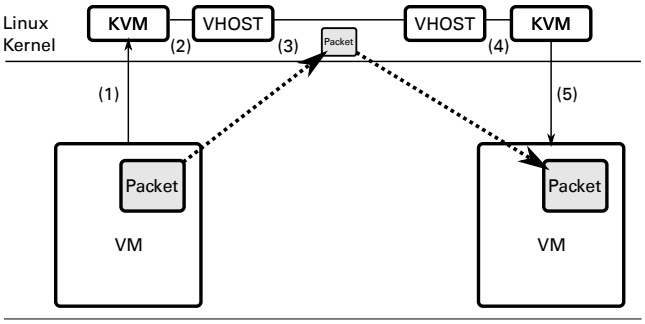

Figure 1. Control flow of packet transmission between two VMs (A and B) with vhost-accelerated network adapters on a single host. Each virtual NIC has a corresponding vhost thread in the host kernel. The sending VM triggers an I/O exit by writing its `NOTIFY` register (1), which KVM signals to the vhost thread (2). This kernel thread copies the packet data into a kernel buffer and delivers it to the network stack (3). The network subsystem wakes up the receiver's vhost thread, which copies the packet data into a packet buffer of the receiving VM. If an IRQ is necessary, it signals the receiver's vCPU kernel thread to inject an interrupt (4), which delivers the interrupt to the guest (5). No Qemu code executes for packet transmission.

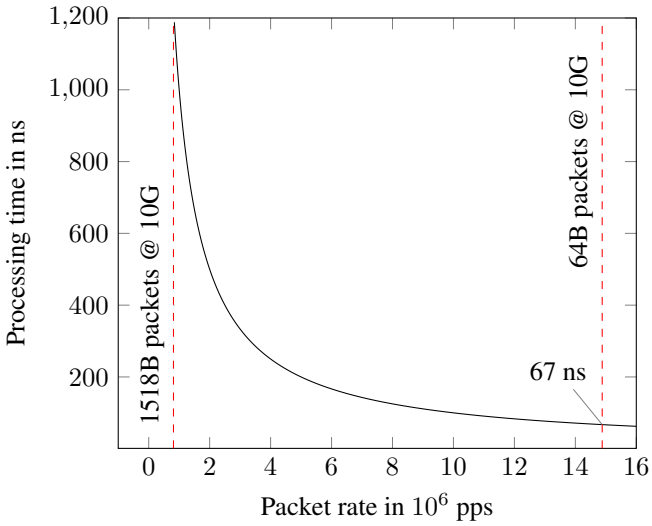

Figure 2. Time available for packet processing for given packet rates on 10 Gbit/s Ethernet. A 64 byte packet must be processed in 67 ns to keep up with line rate. System calls or mutex acquisitions per packet are infeasible.

tiple event file descriptors (`eventfds`) to connect KVM and `vhost-net` as shown in Figure 1. An `eventfd` is a file descriptor that can be used to wait for events. In its most basic form, a read on an `eventfd` will block, until the `eventfd` is written to.

To process buffers enqueued by the guest in a timely fashion, `vhost-net` needs to be informed when the guest wrote the `NOTIFY` register. For this reason `vhost` creates a thread per `virtio` device, which blocks on an `eventfd`. This file descriptor is triggered by KVM, when the `NOTIFY` register is written.

When packets have been delivered, `vhost` needs to inject interrupts into the guest. Interrupt injection happens by binding an `eventfd` to an interrupt source in KVM and giving this `eventfd` to `vhost`, which can trigger it.

The actual setup of these event file descriptors is handled by Qemu in userspace. As of Linux 3.10 when running a network benchmark in a properly configured Qemu/KVM, Qemu itself is not involved beyond the initial setup of the VM.

3. sv3: Packet Switch for VMs

During the development of the sv3 switch we strove to solve the following problems.

Per-packet overhead should be minimized, as this will facilitate throughput even when data is arriving in small packets. As seen in Figure 2, in the extreme case of 64 byte packets, only 67 ns are available per packet on average to keep up line rate. While achieving 10 Gbit/s line rate with small packets on commodity hardware certainly remains a challenge, it is clear that heavy-weight operations per packet, such as lock acquisitions or system calls, must be avoided to come close to that goal.

Related to per-packet overhead is the issue of dealing with high notification overhead. We initially believed that our userspace switch will have significant overhead for notifications and IRQ injecting into its client VMs. While this overhead turned out to be less pronounced than we expected (Figure 5), it drove us to abandon the thread-per-VM model of `vhost-net`.

Memory-bandwidth starts to become a limiting factor as network speeds approach the same order of magnitude. For common operations, our switch should thus not copy data needlessly.

3.1 Qemu Modifications

Just as `vhost-net`, we concentrate on the `virtio` network adapter. In `vhost-net`, the `virtio` implementation is split between the code handling the setup in Qemu and the packet handling path in the `vhost-net` module. To avoid this complexity, we strove to implement `virtio` in a single component only. Because sv3 is meant as a switch for virtual machines, it makes sense to closely tie the implementation of the `virtio` NIC to the switch itself.

Qemu offers no facility to implement devices out-of-process, so we enhanced Qemu to allow externally implemented PCI devices. Our modified Qemu is able to connect to another process that implements a specific device using a UNIX domain socket.

UNIX domain sockets allow file descriptors to be transmitted between two processes. The modified Qemu creates the guest's memory as a file on a RAM-backed file system

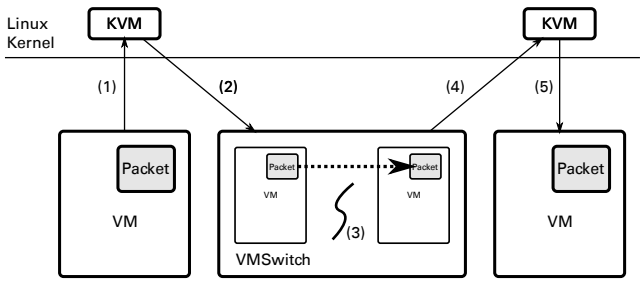

Figure 3. Control flow of packet transmission between two VMs (A and B) with sv3 as packet switch. The sending VM triggers an I/O exit by writing its `NOTIFY` register (1), which KVM signals to sv3 via an `eventfd` (2). sv3 is unblocked by the write to the `eventfd` (3) and copies the packet via its local mappings of VM memory (4). The receiving VM is notified of the packet by signaling an `eventfd` (5) that is bound to KVM's IRQ injection (6). KVM then injects a virtual interrupt (7).

and transmits the corresponding file descriptors via the domain socket. The external process can then `mmap` the guest's memory in its own address space. Further `eventfd`s are exchanged for interrupt injection and I/O notifications.

Our implementation of external PCI devices is complete enough to handle devices with MSI-X interrupts, DMA, and port I/O. This covers `virtio`, but with minor additions, we can implement other PCI devices in external processes as well.

3.2 Implementing `virtio` in sv3

Using our Qemu modifications, we are able move the complete implementation of the `virtio` PCI device into the switch process. This has advantages beyond just simplicity. Because the switch has direct access to `virtio` DMA queues and can communicate directly with KVM for interrupt injection, there is no involvement of Qemu in packet handling.

Uncritical port I/O by the guest on the I/O addresses of the `virtio` device are relayed to the switch via the domain socket connection. This is only used during `virtio` configuration by the guest. Afterwards, the guest only writes to the `NOTIFY` register of the device. The switch instructs Qemu to attach an `eventfd` to this particular register. When a write occurs, KVM will trigger this `eventfd` instead of notifying Qemu of the I/O operation.

3.3 The Switching Loop

The key part in sv3 is its switching loop. Shalev et al. [25] have shown that a single threaded network subsystem is feasible. We build on this result by having a single thread in sv3 to do all packet switching. This thread executes a loop that continuously checks all ports and their queues for work until no port has outgoing packets pending. At this point it will block on an `eventfd`. Writes by a guest to the `NOTIFY` register of a `virtio` device trigger exactly this `eventfd`, thus unblocking the switch.

In effect, there is no need for mutexes and atomic operations. Operations on switch data structures that rarely change and are modifed outside the switching loop, such as the list of attached ports, are serialized with userspace RCU [7].

While the switching loop is active, sv3 disables notifications for all client VMs. In particular, this means that while the switch is busy copying packets for one VM, another VM need not cause a VM exit to notify the switch of activity in its DMA queues. The busier the switching loop gets the less notifications are needed in the system, until there are no notifications needed at all. sv3 only enables notifications from VMs when it is idle and about to sleep.

Depending on the destination, a packet originating from a virtual machine can take two paths. If the destination is another VM, the packet will be directly copied from the source VM's memory to the receiving VM in a run-to-completion fashion. If the receiving VM does not have sufficient network buffers queued at its `virtio` device, the packet will be dropped.

While packets that are destined for a VM are either delivered directly or dropped, packets destined for the physical network adapter are sent asynchronously. sv3 translates the packet description given as `virtio` descriptors into DMA descriptors of the physical NIC. The switching loop then polls regularly to check when the transmission is completed and in turn indicates completion to the sending VM. At this point, the sending VM can reuse its packet buffers. This mode of operation will complete transmit operations potentially out-of-order from the perspective of the guest, a use-case that was already anticipated by Russel [24] in the original `virtio` design to allow zero-copy transmit. Since only packets delivered to different switch ports may be reordered, the performance of TCP connections is not affected.

As already hinted, with all guest memory directly visible in the sv3 process additional packet copies can be avoided. In contrast to the Linux kernel, where large virtually contiguous in-kernel mappings of user memory are problematic, and where user memory is usually handled pagewise, a userspace solution has the luxury of a simple virtual memory layout: `mmap`ing large files into an application's address space is a common operation.

With packet switching implemented in a single thread, sv3 has to do its own scheduling of how long to service each port. The main tunable parameter in sv3 is the number of packets the switch will deliver from a single queue until it considers packets from another queue. The rationale to set this batch size to values larger than one is to exploit the warmed up cache. Large values obviously affect packet latency.

Choosing the optimal batch size is not the scope of this paper. We use a value of 16 by default and note that this can lead to unfair behavior of the switch when different

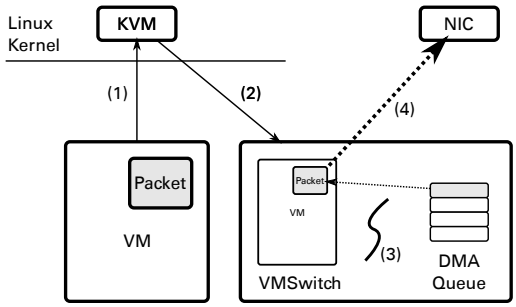

Figure 4. Receiving or sending a packet using the physical network adapter. Step 1 to 3 work as in the VM-to-VM case shown in Figure 3 on the facing page. sv3 translates `virtio`'s packet description to DMA descriptors understood by the NIC. The NIC then reads the packet data directly from the guest's memory.

.

VMs send packets of different average size. In this situation, the switching loop will spend more time servicing the VM sending larger packets.

When latency is of concern, the switching loop can be instructed not to block when the switch is idle, until a certain time has passed. We call this the *idle poll time*. An idle poll time of zero means that the switch will immediately sleep once it has nothing to do. This is the default mode of sv3. A large idle poll time turns the switch into polling mode, where it never sleeps. For the rest of the paper we do not consider poll times other than zero, as it makes the switch behavior hard to compare to `vhost-net`.

The main differences between `vhost-net` and sv3, besides whether they run in the kernel, are their styles of execution. While `vhost` employs one thread per VM, sv3 utilizes a single thread that delivers packets in a run-to-completion fashion.

During overload, i.e. more packets are queued than the network subsystem can handle, sv3 degrades gracefully in that it does not need notifications by VMs anymore, as explained earlier. `vhost` will instead have multiple busy threads that need to synchronize and compete for compute resources with vCPU threads and each other.

3.4 External Connection with Userspace Drivers

Even a packet switch for virtual machines needs an upstream port to transmit packets beyond the boundaries of a single physical host. We decided against using Linux's networking subsystem for this purpose, because it only offers `PF_PACKET` socketsfor high-performance raw packet reception and transmission. We consider `PF_PACKET` sockets limiting, as they require batching of buffer operations and thus introduce unnecessary delays. Additionally, they only support zero-copy transmission from a fixed kernel-provided memory region, which makes direct transmission from VM memory impossible.

We instead decided to drive the physical NIC in sv3 itself, using the `VFIO` framework [30]. `VFIO` is primarily meant for implementing PCI passthrough in Qemu, but is sufficient for implementing drivers for PCI devices in Linux userspace directly.

`VFIO` allows binding event file descriptors to interrupts. In sv3, we use this feature to unblock the switching loop when the NIC signals incoming packets. This is analogous to how VMs unblock the switching loop by writing to their `virtio`'s `NOTIFY` register.

By including the device driver in the switch, zero-copy packet transmission is possible[2] and the NIC sends packets directly from guest memory. Receiving packets requires an additional packet copy, because the NIC delivers packets before the switch knows where to deliver them.

The switch passes stateless offloads programmed by the guest for a packet to the NIC. Given guests and an upstream NIC that all support TCP segmentation and checksum offloads, the switch never needs to segment packets or compute checksums on its own. The switch also passes Large receives, i.e. TCP packets belonging to the same connection that have been merged into a single packet, on to guests, if they support it.

We implemented a driver for the Intel X520 NIC, a popular 10Gbit Ethernet adapter, from scratch. This particular NIC, as practically all competing 10Gbit NICs, offers a superset of the offloads that are possible with `virtio` and is therefore a good fit for sv3.

4. Evaluation

We evaluated our system on an Intel Core i7 3770S CPU running at 3.1 GHz. Memtest86+ [1] reports 18975 MiB/s memory bandwidth (roughly 159 GBit/s). Memory access is uniform. Our host system uses Fedora 19 with a vanilla Linux kernel version 3.10.18 and has power management and frequency scaling disabled. As already mentioned in Section 3.4, we used an Intel X520 network adapter. Our guests used a stripped down 3.10.10 kernel. We used our modified Qemu based on version 1.5.0 for all tests. Disregarding small changes in how guest memory is allocated, our modifications do not touch Qemu's own `virtio-net` implementation.

4.1 Security

One of the major concerns with code executing in kernel space is its security and safety, since one programming error can crash the system, cause unrelated subsystems to fail, disclose information, or in the worst case allow attackers to gain control of the system.

With respect to sv3, there are two sides to this issue. The first one is *attack surface* and the second the implications

[2] Zero-copy packet transmission is only possible of VM memory is locked. If the system overcommits memory, an additional copy is required, because the NIC cannot DMA into guest memory in this case.

of a successful attack. With regards to an attacker that can craft arbitrary packets and fully control one of the connected virtual machines, sv3 shares the same attack surface as KVM used with `vhost-net`, that is the `virtio` interface and those parts of KVM that are necessary for mapping specific VM exits to `eventfds`.

While the latter feature in the KVM module might be superfluous and could be removed from the kernel, if general userspace VM exit handling were fast enough [28], the real value of userspace packet switching lies in the mitigation of attacks.

Because sv3 is an ordinary Linux application, existing hardening techniques, such as sandboxing, can be easily deployed making security relevant flaws harder to exploit for an attacker and attacks themselves easier to detect. Even a successful attack on sv3 will only grant an attacker user privileges on the host, control over the physical network adapter and access to memory of the connected virtual machines.

Access to VM memory can further be restricted by a virtualization layer that emulates an IOMMU [2]. Qemu needs to share only memory that guests mark as DMA-able with a sv3 instance.

Control over the network adapter is also not as valuable as it appears. Due to the use of the host's IOMMU to safely drive the device in userspace in the first place, DMA cannot be used to subvert the rest of the system. The Linux kernel makes sure the application can only establish IOMMU mappings to memory its own memory and an attacker would need to overcome this security feature first.

Finally, sv3 does not need root privileges.[3] It only needs access to its UNIX domain socket and the VFIO device file of the network adapter is supposed to drive. Sandboxing sv3 is trivial.

4.2 Resource Consumption

Low resource consumption is one argument in favor of processes instead of virtual machines as units of disaggregating operating system functionality. Measuring exact memory usage of a Linux process is not straightforward. We measured sv3's memory footprint by observing its *resident set size* (RSS) as displayed by the tool `ps` and subtracted the amount of shared guest memory, as the latter would otherwise be accounted twice, once for the virtual machine and once for sv3. Note that RSS does not include kernel data structures, such as page tables.

Without the upstream network driver sv3 consumes below 2 MiB of resident memory and currently needs 384 bytes metadata per virtual switch port. The Intel X520 NIC driver increases memory usage to 16 MiB, mostly because of a liberal amount of packet buffers.

[3] Root privileges *are* required for setting interrupt affinity, if that is required, because the VFIO interface is incomplete in that respect. Privileges can be dropped after initialization and before a VM is connected to the switch. We consider adding this capability to VFIO.

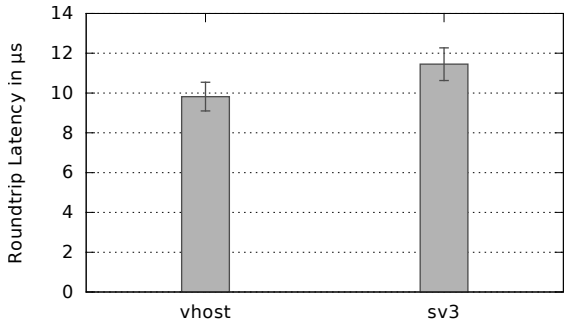

Figure 5. Time between triggering network processing in the guest and receiving an IRQ in the guest for `vhost-net` and sv3 **without** network processing. The overhead for sv3 is caused by traversing the system call layer and switching to userspace. Error bars indicate standard deviation.

As a point of comparison, consider recent work by Colp et al. [5], who decompose the privileged Domain 0 in a Xen system by running Domain 0 subsystems in virtual machines. The network backend VM in their systems is allocated 128 MiB of RAM. The smallest VMs in their design use 32 MiB RAM and are confined, unlike sv3, to a very limited programming environment.

4.3 Microbenchmarks

sv3 reimplements a kernel subsystem in userspace. Thus it must use userspace APIs to achieve what `vhost-net` can do with potentially more optimized in-kernel APIs. In this section, we want to measure the direct overhead of using userspace APIs.

In our particular case, sv3 uses `eventfds` to be notified of activity in the guest's DMA queues and to inject IRQs. `vhost-net` uses the same functionality inside the Linux kernel directly and can avoid the kernel to user mode transition.

We modified both implementations to not do any packet processing, but inject an interrupt directly after being notified by the guest. In Figure 5, we measure the time that elapses between triggering network processing from the guest and receiving the IRQ in the same virtual machine. This time includes, for both systems, the same overhead for the actual VM exit and interrupt injection done by KVM. sv3 adds overhead for traversing the system call layer and switching to and from userspace.

The additional overhead is below 2 μs. The design decision to minimize wake ups by using only one switching thread, may thus not be as important as we initially thought.

4.4 Performance

We evaluate the efficiency of sv3 by measuring the CPU utilization of the host system for constant-throughput TCP streams and packet latency. TCP stream measurements were generated using `nuttcp` [20]. `netperf` [19] was used to assess latency using its request/response benchmarks. We

compare sv3 to `vhost-net`, which had its zero-copy mode enabled.

We consider two basic scenarios. In the first scenario, an external machine generates the load and a VM on the test system receives it. We use the userspace driver of our Intel X520 NIC, which is built into sv3. `vhost-net` uses the normal Linux network stack and thus the Linux driver. We tried to use hardware features, such as Large Receive Offload and interrupt rate throttling, in our driver in the same way as the Linux driver to avoid distorting the results.

We throttled the interrupt rate to 10000 interrupts per second for both `vhost-net` and sv3. This value corresponds to the "bulk latency" setting in Linux' ixgbe driver.

The second scenario moves the load generator into a virtual machine. The traffic stream is thus between two VMs on a single host. No external traffic is involved. Because `nuttcp` consumes a full core for sending constant-throughput streams, we do not show CPU utilization in this case, because it is always equal to one.

External-to-VM Traffic Figure 6 on the next page shows CPU utilization for TCP streams from the external source with TCP segmentation and large receive offload enabled (left) and disabled (right). With both offloads enabled, `nuttcp` produced stable results upto 8 Gbit/s. As the receiver does not fully consume its core, the sender must be the bottleneck.

At 8 Gbit/s, sv3 uses 30 % of its core. The CPU utilization for both systems is almost identical. The reason is that both systems use the same architecture to receive packets. The network adapter copies packets into anonymous buffers. After receiving the IRQ, the Linux kernel wakes up the networking subsystem or in the case of sv3 unblocks an `eventfd`. The thread that is responsible wakes up and copies the packet to the guest VM's memory. Afterwards, the guest is notified of their arrival. The only difference is that sv3 uses `eventfd`s to be notified by the kernel and to notify the guest.

The right part of Figure 6 shows the same setting, but with TCP segmentation and large receive offloads disabled to put more stress on the network path. Instead of large packets the network adapter will now only deliver MTU-sized packets. We used a standard MTU of 1500 bytes. We did not measure `vhost-net` performance, as we could not disable large receive offload. As with the measurements with offloads enabled, the receive is the bottleneck. At 6 Gbit/s, sv3 uses 55 % of its core.

VM-to-VM Traffic Throughput results for our second scenario, VM-to-VM traffic, are shown in Figure 7 on the following page. With offloads enabled (left), we get stable results with `nuttcp` up to about 30 Gbit/s. For throughput below 10 Gbit/s, sv3 is slightly more efficient compared to `vhost-net`. With throughput beyond 10 Gbit/s, this effect is more pronounced. Between 12 and 13 Gbit/s the receiving VM starts to fully utilize its core and essentially goes into polling mode, which is reflected in the curve.

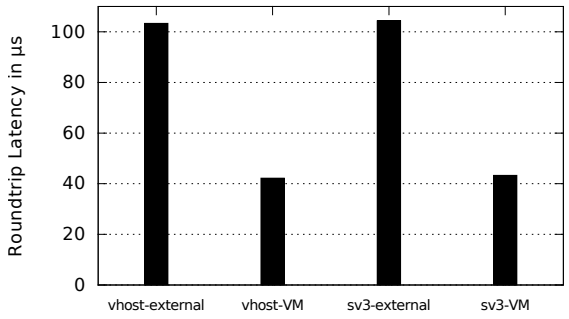

Figure 8. Roundtrip latency for TCP packets between two virtual machines (VM), a virtual machine and an external load generator (external). Standard deviation is below 0.4μs for all measurements.

The right diagram in Figure 7 shows VM-to-VM traffic with offloads disabled. `vhost-net` is not able to reach 7 Gbit/s, because it fully utilizes all four cores[4]. For sv3, virtual machines are the bottleneck at 9 Gbit/s as the switching loop consumes only one core at 50%.

Above 7 Gbit/s, when the guest starts to fully utilize its core, sv3's CPU utilization drops. We attribute this to additional batching and less frequent wakeups. The single-threaded model with little per-packet overhead seems especially suited to this scenario.

Latency Judging from our microbenchmarks, we anticipated a modest increase in latency by using sv3 for external traffic and expected a slight decrease in the VM-to-VM case for sv3, because the packets are handled by only one thread instead of two.

Latency results are shown in Figure 8. We observe that with both systems latency is practically identical. We see an increase of roughly one microsecond for sv3 compared to `vhost-net`.

5. Discussion

5.1 Userspace Switching vs. Driver Domains

Our design shares aspects with the Driver Domain model as it is used in Xen [17]. In particular, it isolates packet processing in a separate component, which is in our case a process in the host system and in Xen's case a dedicated virtual machine.

The overhead of a dedicated virtual machine has lead to architectures, such as Hyper-Switch [21], that optimize network performance by moving the data plane into the hypervisor and leaving only the control plane isolated.

We argue that processes offer a lighter-weight abstraction than virtual machines, which still allows the full soft-

[4] As explained earlier, the sending VM's utilization is not shown, because it is always fully utilizing one core.

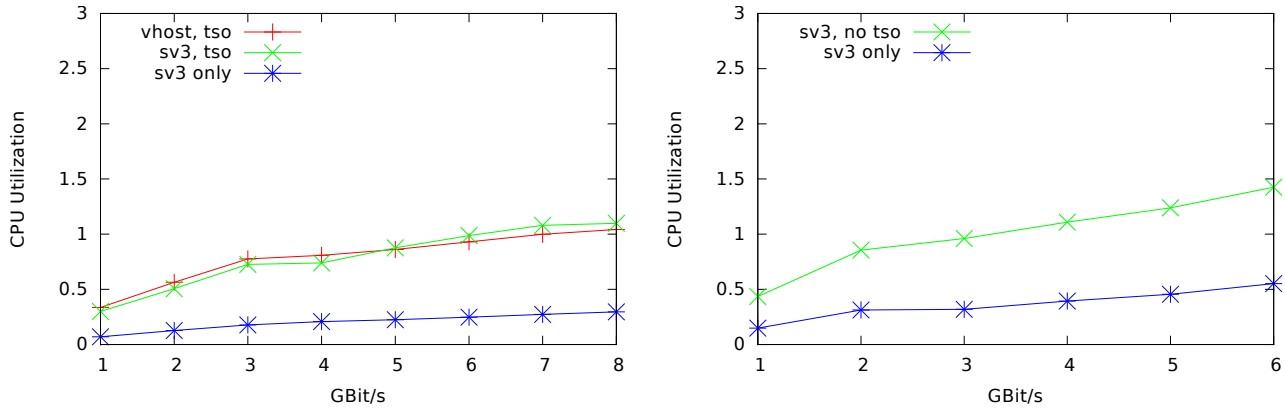

Figure 6. CPU utilization for receiving a TCP stream from an external source. We measured utilization of the whole system, except for the switch-only measurement, which only shows CPU utilization of the software switch itself. The left diagram shows the default configuration with all offloads enabled, the right measurement was done with TSO and LRO disabled. The Linux driver did not support disabling LRO, so it is not shown in the right diagram.

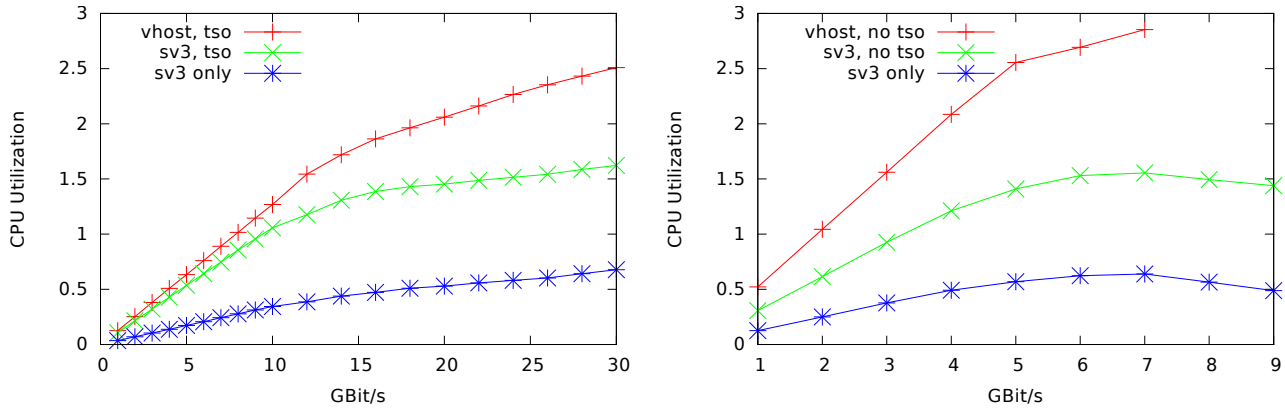

Figure 7. CPU utilization for receiving a TCP stream from another VM. CPU utilization of the load generator VM is excluded. The maximum load of the `vhost-net` configuration is three, when the receiver's vCPU thread and both vhost threads are compute bound. For sv3, the maximum load is two.

ware switch to run isolated from the hypervisor without special privileges. We detail sv3's memory consumption in Section 4.2.

5.2 Multiserver Operating Systems and SMP

One of the reasons for implementing performance critical functionality in the kernel (or hypervisor) is performance. In a monolithic kernel, components can simply call each other with a function call. Yet the example of `vhost-net` in Figure 1 shows that in realistic scenarios multiple threads interact through asynchronous communication even in a monolithic kernel.

On a modern system with a multicore CPU, each of those threads will execute on a different core, unless the system is overcommitted. Frequent context switches between different address spaces do not happen. In the ideal case, each core executes a single thread in a single address space. The cost of switching an address space becomes meaningless to the overall performance of the system. Isolating subsystems in separate address spaces is practically for free. The performance difference of components running in kernelspace compared to userspace now depends on the efficiency of asynchronous notifications.

By consolidating work onto a single thread, our network switch tries to stay active as long as possible compared to the thread-per-VM model employed by `vhost-net`. sv3 avoids the need for frequent notifications when possible and thus mitigates even this source of potential inefficiency. In return, we get a high-performance subsystem running in userspace.

We believe that this reasoning also applies to other components that are typically implemented in a monolithic kernel. A case in point is the use of asynchronous system calls as in the FlexSC system [27], where system calls are used via shared memory and asynchronous notifications. sv3 can

be considered an application of this idea with the twist that our "system call handler" is not running in the kernel.

The design of our userspace network switch is applicable to and inspired by microkernel-based multiserver systems [9, 15, 28].

5.3 A Monolith in Userspace

It seems as if we are just exchanging a monolithic kernelspace component with a monolithic userspace component, but there are important differences compared to the kernel solution.

sv3 can be started multiple times, each time creating an isolated network switch. Errors in sv3 do not propagate to other instances assuming that the underlying kernel is correct. More importantly failures in sv3 do not interfere with kernel subsystems.

The robustness of a single sv3 instance can, of course, be increased. We currently run NIC drivers in the context of the switching loop. A malfunctioning device or bugs in the driver code can potentially wreak havoc on all connected VMs. By making the driver a separate process that speaks the same interfaces as VMs, this scenario can be avoided at the expense of performance.

6. Future Work

6.1 Scalability and NUMA

As demonstrated in Section 4.4, sv3 compares favorably to `vhost-net` for VM-to-VM and external-to-VM communication, because those workloads rarely cause the switching loop itself to be the bottleneck. Especially for high packet rates, such as the measurement in Figure 7 on the facing page with TCP segmentation disabled, the single-threaded run-to-completion approach of sv3 clearly outperforms the model where one virtual NIC is driven by one thread as in `vhost-net`.

For multiple senders, `vhost-net` eventually scales better, because it can distribute packet processing to different cores, whereas sv3 in its current design faces a wall when packet processing chokes a single core, even when virtual network throughput is far from the theoretical maximum dictated by the memory bandwidth of the system. Another issue is that systems with NUMA configurations might exhibit poor performance, if sv3 needs to read and write packet data from distant memory.

One idea to address scalability is to start multiple sv3 instances, one per NUMA domain. VMs are connected to sv3 instances with the same NUMA affinity and, depending on the system, share a last-level cache. Each sv3 instance would still be able to switch packets on the same NUMA node without additional copies. The sv3 instances themselves are connected to each other and use a shared memory region for each connection to forward packets destined to "remote" virtual machines.

Advantages of such a configuration are that performance scales with the number of NUMA nodes, without introducing complexity in sv3 in the form of multithreading. Additionally, service disruption caused by a crash of one sv3 instance is confined to a handful of VMs.

6.2 Integrating Advanced Interconnects

Cui et al. [6] propose to offer virtual TCP offload engines to VMs and use advanced interconnects, such as InfiniBand, to optimize the networking performance between VMs on different hosts. The implications of this approach are appealing, because what a VM initiates as a TCP connection can be mapped to the most efficient network technology or protocol between two virtual machine hosts, if these hosts cooperate. Instead of speaking TCP between two VMs on the same host, plain `memcpy` with primitive congestion control can be used. For remote connections, data transfer can be implemented using RDMA. The transport mechanism can even be transparently switched, if VMs are migrated. We plan to integrate such functionality in sv3.

7. Related Work

Our paper touches the areas of I/O scalability in the context of virtualization, network stack performance optimizations and operating system design. Additionally, it relates to OS and hypervisor disaggregation efforts.

VALE [23] is a software switch for virtual machines that achieves impressive packet forwarding rates. It reduces per-packet overhead by using a more efficient interface to transmit packets [22], removing VM exits, batching, and the reduction of mutex acquisitions. VALE is implemented in the kernel and shares the same security considerations as the standard `vhost-net` networking path.

Ram et al. [21] present Hyper-Switch, a scalable software switch for virtual machines with a split design to reduce the TCB, which we have already discussed in Section 5.1.

Shalev et al. [25] move the TCP/IP stack of a general purpose operating system onto a dedicated core. The isolated stack (IsoStack) uses queues to communicate with the rest of the system. We adopted design ideas of IsoStack for sv3, specifically the idea to have a main loop that serves all clients, is lockless, and does not need atomic instructions. sv3 shares the major drawbacks with IsoStack. In such a design, it is hard to scale network processing to two or more cores.

With their system ELVIS, Gordon et al. [10] try to remove VM exit and interrupt generation overhead by modifying `vhost-net` to run a single thread polling all guests similar to sv3. ELVIS runs in kernel space and is effective at reducing exits, but fully utilizes a core even for light load. The authors present a heuristic to distinguish latency-sensitive from throughput-oriented workloads that is applicable to sv3 as well. Given the experiences from building sv3, we believe

that it is possible to port or reimplement ELVIS in userspace with similar performance to the in-kernel version.

SnabbSwitch [26] is a userspace packet switching and routing framework written in Lua. Its intent is to enable the rapid creation of software-defined networking functionality running on Linux in userspace. Our modifications to Qemu are applicable to SnabbSwitch and may improve its performance significantly.

With their Factored Operating Systems, Wentzlaff and Agarwal [32] rethink the operating system for a many-core system as fleets of servers, where each fleet implements an operating system service. The individual servers in a fleet are bound to particular processing cores and do not directly compete with applications for hardware resources. A future version of sv3 consisting of multiple cooperating instances may be seen as an incarnation of such a service fleet.

Minix 3 [11] uses userspace drivers and components and argues that restartable system components can greatly enhance the reliability of the whole system. Especially, components without or with little state should be easily restartable. sv3 falls into the latter category.

Userspace drivers have been shown to have negligible overhead compared to drivers running in the kernel [16]. Since then, the interfaces for userspace drivers in Linux have been vastly improved [30]. Our switch uses these interfaces.

Hruby et al. [12] agree that dedicating cores to I/O functionality is worthwhile and rearchitect a TCP/IP stack by putting the different layers onto different cores. In contrast to sv3, which follows the run-to-completion philosophy, each networking stack layer is a single threaded program. The authors argue that this is a good design choice for robustness as the layers themselves cannot include concurrency bugs and each layer can be restarted individually.

Colp et al. [5] decompose Xen's monolithic and privileged Domain 0 into several virtual machines according to the principle of least authority. sv3 achieves the same for the networking path in KVM with processes instead of virtual machines with a much lower memory footprint.

8. Conclusion

We have shown that it is possible to reimplement high performance components from an existing monolithic kernel in an unprivileged user process without sacrificing performance. sv3, our userspace network switch for virtual machines, replaces Linux' `vhost-net` subsystem, the in-kernel packet switch, and the device driver for the upstream network adapter. It relies only on basic virtualization functions in the Linux kernel, such as event forwarding and IRQ injection via event file descriptors. No networking code needs to run in privileged mode to achieve high performance networking.

By tightly coupling NIC drivers with the switch, we see identical CPU utilization compared to the in-kernel networking path and negligable latency increase for external-to-VM traffic. VM-to-VM traffic is handled particularly efficient in our implementation, especially at high packet rates, because we reduce per-packet overhead with our design.

The limiting factor of sv3 is that it cannot yet utilize multiple cores for packet processing. For a large number of VMs and sufficient computational resources, it will eventually be outperformed by systems, such as `vhost-net` that use one thread per VM. However, given our microbenchmarks, we are confident that a thread-per-VM design can be adopted for a userspace switch as well.

The source code of sv3 and our Qemu patches are available at `https://os.inf.tu-dresden.de/~jsteckli/sv3.html`.

Acknowledgments

The author would like to thank Björn Döbel for invaluable input on drafts of this paper and Luke Gorrie and the snabb.co team for the occasional dose of motivation and time spent discussing the ideas presented in this paper.

References

[1] Memtest86+ - an advanced memory diagnostic tool. URL `http://www.memtest.org/`.

[2] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster. viommu: Efficient iommu emulation. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2002181.2002187`.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1247360.1247401`.

[4] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0483-2. . URL `http://doi.acm.org/10.1145/1879141.1879175`.

[5] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 189–202, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. . URL `http://doi.acm.org/10.1145/2043556.2043575`.

[6] Z. Cui, P. G. Bridges, J. R. Lange, and P. A. Dinda. Virtual TCP offload: optimizing ethernet overlay performance on advanced interconnects. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, HPDC '13, pages 49–60, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1910-2. . URL `http://doi.acm.org/10.1145/2462902.2462912`.

[7] M. Desnoyers, P. McKenney, A. Stern, M. Dagenais, and J. Walpole. User-level implementations of read-copy update.

Parallel and Distributed Systems, IEEE Transactions on, 23 (2):375–382, 2012. ISSN 1045-9219. .

[8] K. Elphinstone and G. Heiser. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 133–150, Farmington, PA, USA, November 2013.

[9] genode. Genode operating system framework. URL `http://www.genode.org/`.

[10] A. Gordon, N. Har'El, A. Landau, M. Ben-Yehuda, and A. Traeger. Towards exitless and efficient paravirtual i/o. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 10:1–10:6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1448-0. . URL `http://doi.acm.org/10.1145/2367589.2367593`.

[11] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, July 2006. ISSN 0163-5980. . URL `http://doi.acm.org/10.1145/1151374.1151391`.

[12] T. Hruby, D. Vogt, H. Bos, and A. S. Tanenbaum. Keep net working - on a dependable and fast networking stack. In *Proceedings of Dependable Systems and Networks (DSN 2012)*, Boston, MA, June 2012.

[13] A. Kantee. Rump file systems: kernel code reborn. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 15–15, Berkeley, CA, USA, 2009. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855807.1855822`.

[14] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[15] A. Lackorzynski and A. Warg. Taming subsystems: capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, pages 25–30, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-464-5. . URL `http://doi.acm.org/10.1145/1519130.1519135`.

[16] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, 2005. ISSN 1000-9000. . URL `http://dx.doi.org/10.1007/s11390-005-0654-4`.

[17] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1267359.1267361`.

[18] J. Nakajima. Enabling optimized interrupt/APIC virtualization in KVM. In *KVM Forum*, 2012.

[19] netperf. netperf. URL `http://www.netperf.org/`.

[20] nuttcp. nuttcp network performance measurement tool. URL `https://www.nuttcp.net/`.

[21] K. K. Ram, A. L. Cox, M. Chadha, and S. Rixner. Hyper-Switch: A scalable software virtual switching architecture. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, USENIX ATC'13, Berkeley, CA, USA, 2013. USENIX Association.

[22] L. Rizzo. Netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2342821.2342830`.

[23] L. Rizzo and G. Lettieri. VALE, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1775-7. . URL `http://doi.acm.org/10.1145/2413176.2413185`.

[24] R. Russel. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[25] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIX ATC'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855840.1855845`.

[26] snabb. Snabbswitch. URL `https://github.com/SnabbCo/snabbswitch/wiki`.

[27] L. Soares and M. Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1924943.1924946`.

[28] U. Steinberg and B. Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. . URL `http://doi.acm.org/10.1145/1755913.1755935`.

[29] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005. ISSN 0018-9162. . URL `http://dx.doi.org/10.1109/MC.2005.163`.

[30] vfio. VFIO driver: Non-privileged user level pci drivers, 2010. URL `http://lwn.net/Articles/391459/`.

[31] G. Wang and T. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, 2010. .

[32] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, Apr. 2009. ISSN 0163-5980. . URL `http://doi.acm.org/10.1145/1531793.1531805`.