Plugging the Hypervisor Abstraction Leaks Caused by Virtual Networking

Alex Landau
IBM Research–Haifa
lalex@il.ibm.com

David Hadas
IBM Research–Haifa
davidh@il.ibm.com

Muli Ben-Yehuda
IBM Research–Haifa
muli@il.ibm.com

ABSTRACT

Virtual machines are of very little use if they cannot access the underlying physical network. Virtualizing the network has traditionally been considered a challenge best met by such network-centric measures as VLANs, implemented by switches. We begin by arguing that network virtualization is best done by hypervisors, not switches. We then show that modern hypervisors do a poor job in virtualizing the network, leaking details of the physical network into virtual machines. For example, IP addresses used across the host's physical network, are exposed to guest virtual machines. We then propose a method for plugging the network-related leaks by ensuring that the virtual network traffic is encapsulated inside a host envelope prior to transmission across the underlying physical network. In order to overcome the performance hit related to traffic encapsulation, we analyze the unique case of virtual machine traffic encapsulation, exploring the problems arising from dual networking stacks — the guest's and the host's. Using a number of simple optimizations, we show how an unmodified guest under the KVM hypervisor can reach throughput of 5.5Gbps for TCP and 6.6Gbps for UDP for encapsulated traffic, compared to 280Mbps and 510Mbps respectively when using the default guest and host networking stacks.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.4.4 [**Operating Systems**]: Communications Management—*Network communication*

Keywords

Network abstraction, Network virtualization, I/O virtualization

1. INTRODUCTION

Modern hypervisors are expected to support advanced administrative actions such as Checkpoint/Restart [24] where a guest is paused for a potentially very long period of time, VM Cloning [14] where a guest is duplicated, and Live Migration [5, 20], where a virtual machine is transferred from a source hosting system to a destination hosting system. In order to support such advanced functions, the hypervisor must ensure that the guest application and operating system are able to continue running unchanged even when the compute, network and storage environments in which the guest is running change drastically. This can be achieved by completely abstracting the environment offered to guests and ensuring that guests become unaware of any characteristic of the hosting platform. Such an abstraction was not required by older generations of hypervisors.

Hypervisors, first developed in the mid-60s as part of the IBM 360/67 mainframe computer, were initially designed to create a complete virtual replica of the hosting system [6]. Guest virtual machines were offered a fully protected and isolated copy of the underlying physical host hardware. The hypervisor ensured that the guest operating system and applications would behave exactly as they would on the original hardware [16, 17]. Since the hypervisors traditional role was to serve uninterrupted guests on a given hardware platform, there was no need to completely abstract away the platform environment from the guest. The need for a complete abstraction layer emerged later with the introduction of more advanced administrative actions.

A hypervisor offering networking services to its guests is required to use optionally unique resources such as names and addresses. Such resources are often local and relate to the hosting platform or to the network system serving the hypervisor. When a hypervisor exposes such direct local references to the guest, it allows the guest operating system and applications to learn and use such direct references to the hypervisor environment. Exposing the guest to local network environment resource references such as IP addresses local to the host environment, is a leak in the hypervisor abstraction layer.

Leaks in the hypervisor abstraction layer may tamper with a future decision to migrate a guest to a remote hypervisor, to clone a guest or to restart a guest at a later time where/when direct references previously learned by the guest may either be unavailable or already allocated to a different guest. As an example, consider a guest learning a local IP address. The guest cannot be cloned, cannot be migrated to a remote host, not served with the same IP subnet, and cannot be safely restarted after a long pause, unknowing if the IP address was not allocated to a different entity.

In order for a hypervisor to enable Live Migration, Checkpoint/Restart and VM Cloning, it should construct a leak-free abstraction layer in which guests: (1) are offered an isolated virtual environment on top of the shared physical environment; (2) remain independent of physical characteristics; And (3) remain independent of physical location. Network services, offered by hypervisors to virtual machines should follow the same principle requirements. The hypervisor should ensure that the guests (1) are offered isolated virtual network environments on top of the shared physical network environment serving the hosting systems; (2) remain independent of physical network characteristics such as topology, protocols, address spaces, etc.; And (3) remain independent of physical location (e.g., a network segment) that serves the hypervisor.

This paper, as its name suggests, is focused on plugging the abstraction layer leaks caused by hypervisors offering virtual networking services to guests. Other, non-networking related examples to abstraction layer leaks exists, but are not discussed in this paper (See for example, constrains related to migrating a guest between an AMD and an Intel platforms [27]). The contributions of this paper are:

- We analyze the network virtualization task from the perspective of a hypervisor abstraction challenge. We highlight the need for hypervisors that offer complete network virtualization and identify leaks caused by current common virtual networking approaches.

- We suggest a virtual networking framework for plugging network related leaks in which hypervisors encapsulate guest virtual machine traffic using an external envelope.

- We identify that in order to advance towards wire speed performance, given the new framework, the guest network stack and the host network stack need to be optimized and cooperate. Such a Dual Stack approach implementing the suggested virtual networking service is discussed. The division of work between the two stacks and the design of an efficient solution to the Dual Stack problem is analyzed.

- We present first performance results of our Dual Stack solution showing two virtual machines communicating at TCP throughput of 5.5Gbps (6.6Gbps for UDP). These results offer a 20 times (12 times for UDP) improvement compared to the default host configuration.

2. LEAKS IN EXISTING VIRTUAL NETWORK MODELS

Most current hypervisors offer incomplete network abstraction, leading to abstraction layer leaks. (See for example: VMware [22], KVM [12], Xen [3], Hyper-V [15]). One common approach when offering network services to guests is to allow the guest a direct access either to a dedicated physical network adapter of the hosting platform or to one or more queue pairs of a shared multi-queue network adapter (See for example [29]). An apparent advantage of the direct access approach is reducing the system overhead and improving the performance of the network services offered to guests [28]. Yet, in order for a guest to perform direct access, the hypervisor exposes the guest to direct references of unique platform hardware resources, which tampers with potential future guest migration, cloning or restart.

Hypervisors that refrain from enabling guest direct access to the platform hardware resources may either emulate a network adapter in software or use a paravirtualized mode in which a frontend driver added to the guest is working against a hypervisor backend [18]. Regardless of the mode used, one common approach is for the backend or emulated adapter, to serve guest packets using a local software bridge or router that optionally connects to the physical network. Using a software bridge/router connecting the guest to the physical network offers the advantage of a flat service in which virtual machines and physical machines act the same and are offered the same network services. Sadly, this is also a major disadvantage of this approach as the physical network is ill designed to support Live Migration, Checkpoint/Restart or VM Cloning. A guest connected to a local network via its hypervisor is exposed to network addresses with spatial meaning that may also be time-bounded. Consider as an example DHCP, where clients lease IP addresses for a given time interval based on locality. Connecting the guest to the physical network via a bridge/router therefore introduces a leak in the hypervisor abstraction layer.

Current hypervisors tend to consider this incomplete abstraction as a network problem rather then a problem of the hypervisor. One suggested solution for migrating a guest, exposed to a network reference with local meaning, includes either restricting the migration to a subset of the hypervisors that are connected to the same network segment, therefore allowing the guest to continue using its acquired local references [1, 2]. Another suggested solution includes synchronizing the guest migration with a timely reorganization of the network to ensure that the chosen destination hypervisor is allowed access to the same network segment [8]. In this paper, the network virtualization problem is analyzed and considered to be a task of the hypervisor. It is suggested that hypervisors should offer complete abstraction and avoid exposing guests to network related leaks such as network addresses with local meaning.

Another common approach hypervisors take to address network virtualization is to utilize a Network Address Translation (NAT) function. Using NAT, a virtual machine may use a fixed virtual address. Packets traveling to/from the virtual machine are converted to include the external physical address of the hosting platform [21]. Note however that received packets continue to include references of peer physical addresses, causing a leak in the hypervisor abstraction layer. Such leak may be evident at the guest application, either running inside the NATed guest or in other nodes referring to this guest over the network. As an example, consider a guest acting as an application client and served with NAT. The packets transmitted by the guest are converted by the NAT function such that they would include the external physical address of the hosting platform. The guest may communicate with other nodes of the same application acting as the application servers. The application servers therefore receive packets that include references to the external physical address of the guest hosting platform. Thus tampering with future migration of the client guest and other administrative actions.

3. A NEW FRAMEWORK FOR NETWORK VIRTUALIZATION

Unlike traditional approaches that seek to solve the network challenge outside of the hypervisor, here we consider the hypervisor abstraction layer as the right place for network virtualization. Under the suggested framework, hypervisors would plug networking related leaks, by introducing traffic encapsulation as a hypervisor service. Using traffic encapsulation, the hypervisor can hide all references of local resources from the guest. Thus the use of traffic encapsulation enables hypervisors to establish a leak free abstraction layer when serving guests. The suggested hypervisor service would encapsulate all guest traffic traveling on external network links.

Frames traveling on external links would therefore include an internal envelope which is the outcome of the guest network stack, as well as an external envelope, added by the hypervisor. The external envelope may include protocol specific fields as well as host specific protocol headers, such as IP and MAC headers to allow packet forwarding between hosts. This means that when guest frames leave the guest network stack and enter the hypervisor, the hypervisor should add an external envelope, prior to sending the frames. Since the external envelope includes standard protocol headers, the hypervisor can be expected to use a hypervisor (or host) network stack.

We suggest considering the guest network stack and the hypervisor network stack, not as independent complete network stacks, but rather as two fragments of a complete Dual Stack solution. The suggested Dual Stack approach is helpful in optimizing the service offered to guests and reducing unnecessary overheads. In order to reach acceptable performance, the two fragments, the guest network stack and the hypervisor network stack, need to be optimized and cooperate such that the work traditionally performed by a single complete stack would now efficiently be divided between the two stack fragments.

One notable constraint on the suggested Dual Stack approach, is that in many cases, the virtual machine is expected to be unaware of being virtualized, limiting the optimizations that can be introduced at the guest network stack fragment. The hypervisor network stack fragment can be more freely optimized.

4. RELATED WORK

When a hypervisor serves layer 2 frames coming to/from virtual machines by encapsulating them and sending them across the host network, the hypervisor is said to create an *Overlay Network* in which a virtual network overlay connecting virtual machines is created on top of the host physical network (the underlay).

Several research groups have investigated the problem area of constructing an overlay network to serve as a virtual network between virtual machines. Project VIOLIN, Virtual Internetworking on OverLay Infrastructure (see [9, 19]) connects virtual machines physically deployed on separate subnets to a virtual Local Area Network. VIOLIN mimics the structure of a standard LAN with physical hosts and switches connected by physical links to serve Virtual machines. The physical links are emulated by using UDP tunneling. The LAN physical switches are emulated by using a single centralized virtual switch service. A UDP tunnel therefore connects each virtual interface of a virtual machine to the respective virtual switch serving the virtual LAN. The virtual interface nodes register to a virtual switch which maintains a centralized routing table and forwards traffic between the nodes. A star topology is used. The use of a centralized virtual switch forces co-located nodes to communicate via a possibly distant virtual switch.

The VNET project uses a similar approach [23]. Virtual Machines are served using VMWare's host-only connection transferring the virtual machine frames to the host. A host service then encapsulates the frames using TCP/SSL and forwards them to a remote virtual switch. VENT extends VIOLIN by allowing the virtual switch to take heuristic routing decisions and ask nodes to establish adaptive direct connections. VNET therefore improves the solution efficiency, but remains dependent upon a centralized switch.

VANs, suggested by [7], offers a distributed alternative. Unlike VIOLIN and VNET, a centralized virtual switch is not used. Instead, a distributed virtual switch is implemented between hosts. The VAN host daemon uses a host UDP port to communicate with peering daemons located on other hosts. Such communication includes both a data plane and a control plane. The control plane includes auto discovery and dynamic routing between peering hosting platforms to reduce the need for complex management. Under KVM, VANs were implemented by combining a TAP device [13] with a centralized traffic encapsulation host service. A TAP device simulates a virtual Ethernet device attached to the host. The use of TAP devices allows the host administrator to freely serve the QEMU guest with different networking services available at the host. VANs suggested that the administrator would connect the TAP device to a centralized VAN daemon. In this way, guest frames forwarded by QEMU via the TAP device would be encapsulated by the centralized VAN daemon before being transmitted across the host network encapsulated within a UDP packet. Similarly, frames received via the host network and destined to guests are decapsulated by the host VAN daemon and transfered via the virtual Ethernet interface to QEMU where they are forwarded to the guest.

The first design of VANs, like VIOLIN and VNET is not geared to wire speed performance. All three solutions make use of host network services and try to solve the network virtualization challenge outside of the hypervisor. Solving the challenge as part of a host service as suggested by VIOLIN, VNET and VANs results in high latency, low performance service to guests. VIOLIN and VNET further make use of a centralized virtual switch resulting in inefficient routes and interfering with the resulting performance offered to virtual machines. Here, we target offering virtual machines with wire speed performance. We therefore assume the VAN distributed switching approach and replace the hypervisor data plane with a new framework designing virtual networking into the hypervisor abstraction layer.

5. TRAFFIC ENCAPSULATION IN KVM

5.1 The Existing Network Path

Under the KVM Hypervisor [12], guests are implemented in the framework of QEMU user space processes and may be offered different host networking services. See figure 1. In order for the QEMU process to utilize the host networking services it is required to communicate layer 2 frames between the guest and the host. QEMU commonly communicates layer 2 frames to/from the host using TAP devices. Using the TAP device, the administrator may connect the guest to the host bridging or routing services and offer the guest

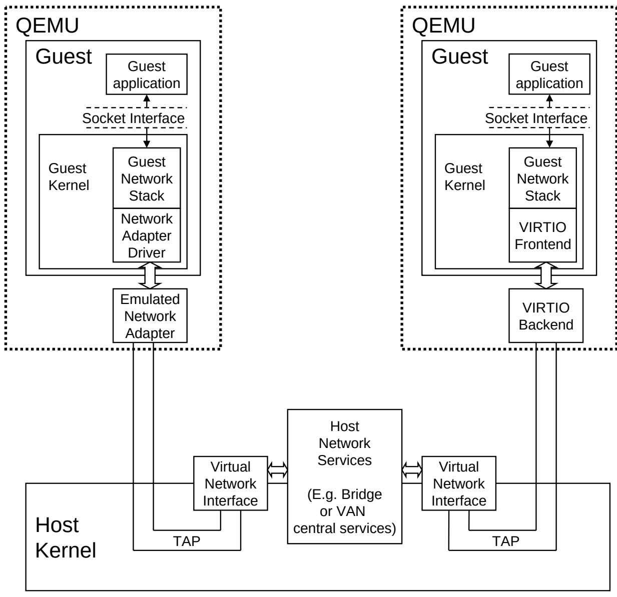

Figure 1: Network services via a TAP Device as used today: A guest application sends and receives traffic via a guest socket interface. The Guest network stack uses a network driver, either a standard one or a paravirtualized frontend driver, to communicate with a QEMU emulated adapter or a backend. The QEMU then uses a TAP device to transfer the traffic to/from the host. The host administrator can connect the virtual network interface of the TAP device to the appropriate host network services.

services such as NAT, DHCP, DNS, etc.

In the case of a fully virtualized mode, the guest operating system uses one of several Ethernet drivers supported by a respective set of emulated network adapters in QEMU. The emulated drivers behave just like their hardware counterparts and expose I/O channels and memory mapped registers which are used by the guest operating system's matching Ethernet drivers. In the case of a paravirtualized mode, the guest operating system uses a frontend network driver that in turn uses a pair of message rings (see virtio [18]) between the guest and QEMU. The frontend is supported by a backend located in QEMU and connected to the other side of the virtio message rings. Recently, it was suggested to move the QEMU virtio backend functionality to kernel space [26].

In both fully virtualized and paravirtualized modes, layer 2 frames transmitted by the guest operating system are transfered to QEMU. When QEMU is configured to use a TAP device, it writes the layer 2 frames to the TAP device. In cases where the host bridge is configured to serve the virtual Ethernet interface of the respective QEMU TAP device, the host bridge would then receive the frames for further processing. The host bridge performs standard layer 2 bridging and may consequently send the frames out to a different bridge interface such as another QEMU instance connected via a TAP device or an external interface connected to the local network.

Frames are received from the host bridge by QEMU using a `select()` and `read()` sequence. Once read, the frames are delivered to the guest operating system either using the backend-to-frontend paravirtualized path or via the emulated Ethernet device and its respective frontend driver using the fully virtualized path.

5.2 A Dual Stack Under KVM

The current research explores a different approach in which encapsulation is performed distributively by each QEMU instance rather than by a centralized host service. See figure 2. The research targets a low latency, high performance networking service for serving guests and to minimize host and guest networking overheads while plugging the hypervisor networking related leaks. Here, layer 2 frames transmitted by the guest operating system via the frontend-backend path or via an emulated network adapter are then encapsulated by the hypervisor and sent out using the standard network stack of the host. Packets received using the host network stack are decapsulated by the hypervisor and forwarded to the guest. Adding encapsulation/decapsulation abilities to QEMU offers an opportunity to improve the performance of encapsulation-based virtual networking services. Packets transmitted by the guest application go through the guest network stack first, then passed to the QEMU hypervisor code which sends them via the host network stack. If and when the QEMU virtio backend moves to kernel space, the encapsulation will follow suit, as doing encapsulation at the backend prevents performance-degrading kernel-user mode switches. This paper seeks to explore the behavior of the dual stack path between guest applications and the host network.

6. TOWARDS MAXIMAL PERFORMANCE

When two network stacks – one at the host and the other at the guest – are involved in the transmission and reception of network packets, it is extremely important to ascertain

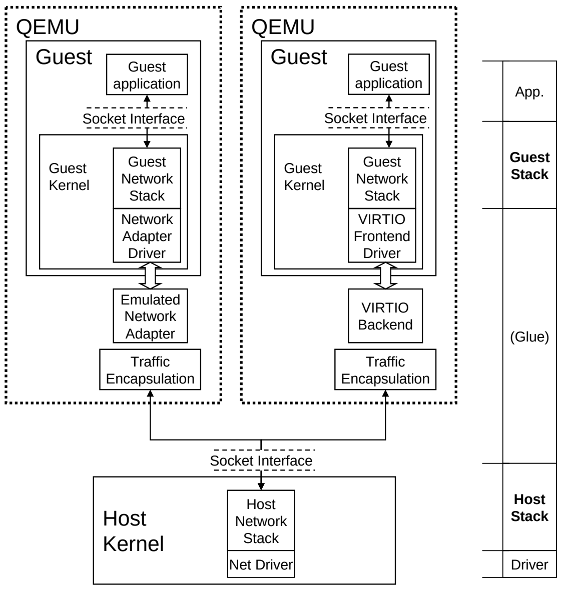

Figure 2: A Dual Stack Hypervisor Service implemented by QEMU: A guest application sends and receives traffic via a guest socket interface. The guest network stack uses a network driver, either a standard one or a paravirtualized frontend driver, to communicate with a QEMU emulated adapter or a backend. The QEMU sends encapsulated traffic or decapsulates received traffic via the host socket interface. The host network stack uses a network driver to communicate across the host network.

that both stacks are tuned to work together to achieve maximal performance. In this section we discuss several aspects immensely affecting performance.

6.1 Large Packets

Applications using standard socket interface may send packets up to 64K. A typical size of an Ethernet MTU is 1500 bytes (although some networks support 9000-byte jumbo frames, they are not ubiquitous). Hence, the operating system normally splits outgoing packets into MTU-sized frames, and transmits each frame separately. The decision where to split the packet into MTU sized frames affects the performance of network stacks. Traditionally, packets are segmented at the transport or network layer which may result in multiple frames traversing the network stack per each sent packet. In order to reduce per frame network stack overheads, modern network stacks postpone segmentation of the packet to later in the network stack. Similarly, modern network stacks seek to reassemble IP packets early at the network stack. The benefit is that only one (big) packet traverses the networking stack, instead of being split in software into many MTU-sized packets each traversing the stack as a separate entity.

Modern NICs have a feature called *segmentation offloading.* Such NICs can receive a packet of up to 64 KB in size from the OS, segment it into MTU-sized frames, add the necessary headers and transmit each frame. Using such modern NICs, when an application asks the OS to send a packet, the packet traverses the stack as a whole unit and only gets segmented before transmitting it on the wire. When segmentation offloading is not used, modern network stacks may use a process called *generic segmentation offloading (GSO)* where segmentation is done in software late in the network stack.

In virtualized environments, there is a higher per-frame processing cost compared to non virtualized environments that is the result of:

- A VM exit caused by the guest OS transmitting the packet over the virtual NIC. This state transition carries much overhead in the form of time during which neither the guest nor the host can execute useful work.
- VM exit handling. This is the code running on the host, analyzing the exit reason and finding the right handler.
- Packet transmission on the host, passing through the host stack.

We enabled segmentation offloading at the virtual NIC exposed to the guest to help reduce not only the guest network stack overhead, but also the number of VM exits and the host network stack overhead. To this end, in the case of a dual stack, the packet route from the guest to the wire is as follows:

1. Guest application sends packets larger than the MTU via the guest socket.
2. Guest Network Stack avoids segmenting the packet since the virtual NIC supports segmentation offloading.
3. Guest virtual NIC transfers the packet as a whole to the host.
4. Host receives the whole, unsegmented packet.

5. Host adds the required encapsulation.
6. Host hands the unsegmented and encapsulated packet to the host network stack.
7. Ideally, Host network stack avoids segmenting the packet until it reaches the physical NIC.
8. Physical NIC does the segmentation in hardware and sends the resultant frames on the wire.

A complementary feature, used on the receive path, is called *large receive offload (LRO)*. When used with a physical NIC, it allows the NIC or the NIC driver to aggregate several incoming packets and pass them to the stack for handling as a single entity. Supporting LRO in the virtual NIC allows a guest to receive a packet of up to 64 KB from the host without the host needing to segment and the guest having to reassemble it.

6.2 TCP and UDP Checksums

As a prerequisite for segmentation offloading, a NIC must be able to compute TCP and/or UDP checksums. As it creates frames for transmission by segmenting the original big packet, it has to add network-layer (IP) and transport-layer (TCP or UDP) headers, which contain a checksum field [4]. Since the virtual NIC transmits frames from the guest to the host running the guest, there is no reason to compute the checksum in the guest and check it in the host. Therefore, we modified KVM to signal the guest driver to fill in a dummy checksum and indicate to the host that checksum of this frame should not be verified. Since the virtual NIC is implemented in software, refraining from computing and verifying checksums saves precious CPU cycles.

6.3 CPU Affinity and Pinning

QEMU, KVM's user-space component, creates a thread called *CPU thread* for each guest virtual CPU, and one additional thread called *IO thread* handling I/O on behalf of the guest [11]. Consider the case when the guest has only one virtual CPU. In this case, QEMU will create two threads. When the host has multiple CPU cores, the Linux process scheduler has the freedom to schedule both threads to run on the same core or on different cores. Moreover, it can move the threads from core to core.

Given that the CPU and IO threads consume more than a single core, scheduling them to run on a single core will limit maximum performance. However, the Linux scheduler was observed to do just that. Moreover, if the host has more than 2 CPU cores, in an attempt to balance workload among cores, the scheduler moves the CPU and IO threads back and forth. This results in cache lines bouncing between cores impeding performance. A performance prone solution is to ensure that the scheduler assigns one fixed core to the CPU thread and a second one to the IO thread. Since fixing the Linux scheduler is out of the scope of this work, we implemented a simple solution that works around both deficiencies by pinning the CPU thread to one core and the IO thread to a different core.

6.4 Flow Control

When using a Dual Stack, care must be taken to avoid duplicating the end-to-end flow control mechanisms. Duplicated flow control introduces unnecessary overhead and may have unwanted effects on the resulting performance. For example when the client application uses a TCP service and therefore activates a sliding window flow control, using a second sliding window mechanism at the host stack, duplicates the TCP sliding window mechanism which may severely degrade TCP performance [25].

End-to-end flow control is the task of the OSI Transport Layer and is not expected to be part of the OSI Data Link Layer. Since guest applications running on a virtual machine are not aware of the underlying host environment, guest applications use whatever end-to-end flow control mechanism suitable for their purposes. At the same time the guest expects no end-to-end flow control to exist as part of its network service. The lower host stack should therefore avoid using an end-to-end flow control mechanism making native UDP a prime candidate for the lower stack network service.

Unlike end-to-end flow control, the lower stack is expected to engage in per link flow control. When two guests reside at two different hosting platforms connected via a network, the guest sender packets are delivered by the sender hosting platform via the host network stack and from there via the physical network links to the receiving hosting platform. This network path is identical to the one used by host applications such that the same per link flow control is used across the different logical and physical links. However when two guests reside on the same hosting platform, the guest sender packets are delivered by the host to the receiving guest via the host network stack and via the host loopback device. An advantage therefore arises in implementing a link flow control across the loopback interface to ensure that the host network stack would stop sending packets across the loopback interface when its receive path to the receiving guest is full.

Note that in order to optimize the network service performance offered to guests, care must be taken to allocate sufficient buffering at the lower host stack to ensure that packets are not lost during transient host conditions. High CPU load on the host may temporarily reduce the number of packets that the host is able to process. This translates into traffic burstiness even when the sender guest sends out traffic with a fixed rate. On top of that, the current virtio implementation transmits packets in bursts. Thus even if the receiver processing speed equals the sender packet producing speed, the host buffers must accommodate the traffic burstiness caused by CPU and implementation. Using insufficient buffers, the traffic burstiness would cause sporadic packet loss, not related to the end-to-end flow control used by the guest sender.

A guest transmitting UDP packets (UDP in UDP encapsulation) encountering a sporadic packet loss would achieve low throughput. Results would become significantly worse for a guest sending TCP packets (TCP in UDP encapsulation). TCP will detect the loss and reduce its transmission speed aggressively. We enlarged the transmit and receive UDP buffers in the host stack to the size of the data in flight, which in the TCP case is the end-to-end bandwidth-delay product, thereby reducing the effect of implementation and CPU related burstiness.

7. PERFORMANCE RESULTS

To quantify the performance of the dual stack implementation under KVM, we started two virtual machines (A and B) with one virtual CPU each on the same host. The host is an Intel Core 2 Quad processor at 2.4GHz with 4GB of

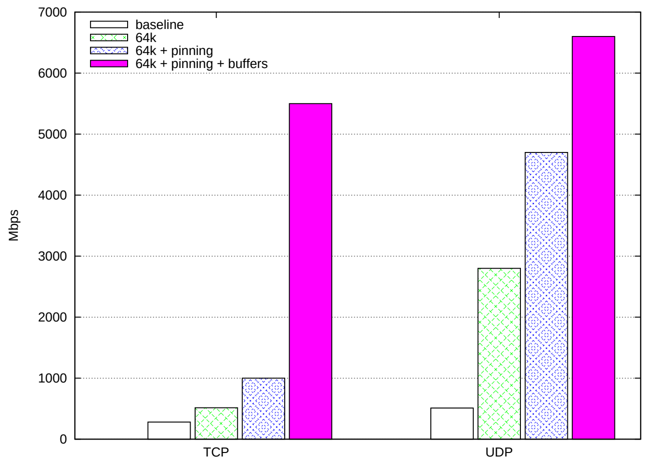

Figure 3: Throughput enhancements for TCP and UDP

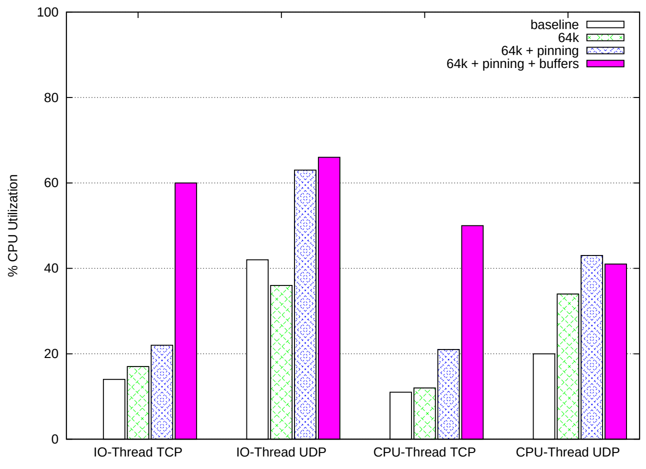

Figure 4: CPU utilization of *sender* QEMU as measured by the host

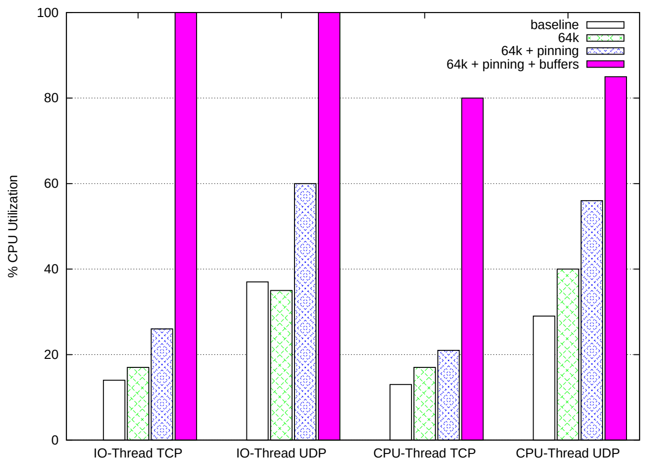

Figure 5: CPU utilization of *receiver* QEMU as measured by the host

RAM. Network frames transmitted by VM A are encapsulated by QEMU into UDP packets and transmitted via a socket on the local host loopback interface to the QEMU process running VM B, which then extracts the packet and passes it to VM B. Loopback, as opposed to a physical NIC, was chosen so as to inhibit the adverse performance effect of managing a hardware device. In this section we show the performance results for this setup, considering the optimization opportunities observed in Section 6.

In the following benchmarks, throughput was measured with NetPerf [10] and CPU utilization with TOP. The performance baseline we compare our results to is when no optimizations are in place, i.e., when guest A sends MTU-sized packets to guest B, no threads are pinned to particular CPUs and the default UDP buffer sizes are used. This is marked as "baseline" in the following figures. The first enhancement we implement is large packets, as described in Section 6.1, marked as "64k". The benchmark was run with the `-m 64k` option to Netperf. The next enhancement is CPU pinning (see Section 6.3). In addition to using large packets VM A's I/O and CPU thread were pinned to cores 1 and 2 respectively, while VM B's I/O and CPU thread were pinned to cores 3 and 4. This was done by using the `taskset` command. The results are marked "64k + pinning". Finally, after increasing UDP buffer sizes (as indicated in Section 6.4) by writing the value 16777216 (16 Mbytes) to the files `/proc/sys/net/core/{r,w}mem_{max,default}`, UDP packets stop being lost on the host, the guest TCP stops throttling its transmit rate, and we get results marked "64k + pinning + buffers".

In Figures 3–6 we show benchmark results for throughput, sender and receiver QEMU CPU utilization, and CPU utilization as measured by the guests. CPU utilization is measured in percents of a single core, while throughput is measured in Mbps by the guest (i.e., this is the "data" throughput; if we count the raw bytes transmitted on the loopback interface including the encapsulation overhead, the figures would be higher).

In Figure 3 we see that the TCP throughput doubles with each new optimization, except the last – increased host UDP buffers – which increases the throughput more than five-fold to a value of 5.5Gbps. UDP throughput quadruples when 64k packets are being used, and reaches 6.6Gbps with all optimizations.

Figure 4 depicts sender QEMU CPU utilization as measured by the host. Both I/O and CPU QEMU threads are shown. In all cases, the utilization will all optimizations is higher than for the baseline. However, the maximum is 66% of a core. The receiver QEMU CPU utilization, shown in Figure 5, suggests that the bottleneck is on the receiver side. With all optimizations in place, the receiver I/O thread's CPU utilization is 100% for both TCP and UDP.

Finally, in Figure 6 we show CPU utilization as measured inside the guests. Interestingly, this time the sender is the one that approaches 100% when all optimizations are in place.

8. CONCLUSIONS

In this paper we identified abstraction layer leaks created by hypervisors offering virtual networking services to guests. We discussed the creation of an efficient Dual Stack solution to achieve network encapsulation and to plug the abstrac-

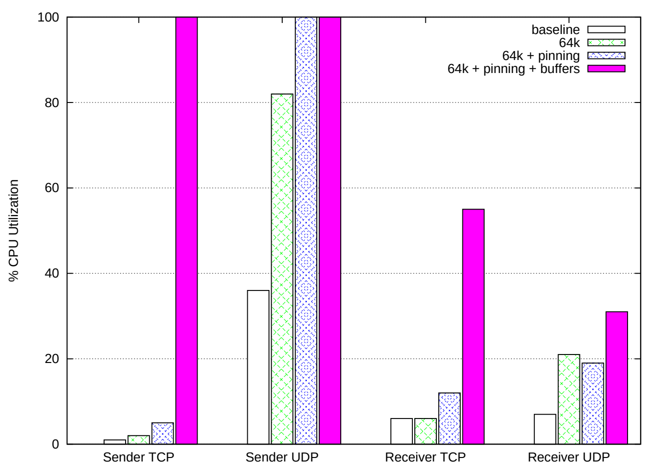

Figure 6: CPU utilization of sender and receiver as measured by guests

tion layer leaks. Finally, we identified and fixed bottlenecks in the performance of the suggested Dual Stack network virtualization approach. We showed that using the new framework, virtual machines can be served with throughputs exceeding 5.5Gbps.

9. ACKNOWLEDGMENTS

The research leading to these results is partially supported by the European Community's Seventh Framework Programme ([FP7/2001-2013]) under grant agreement number 215605

References

[1] *Migration – KVM*. `http://www.linux-kvm.org/page/Migration`.

[2] *Xen 3.0 Virtualization User Guide*. `http://www.linuxtopia.org/online_books/linux_virtualization/xen_3.0_user_guide/index.html`.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[4] R. Braden, D. Borman, and C. Partridge. Computing the Internet checksum. RFC 1071, Internet Engineering Task Force, Sept. 1988.

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[6] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM IBM Journal of Research and Development*, 25(5):483–490, 1981.

[7] D. Hadas, S. Guenender, and B. Rochwerger. Virtual Network Services For Federated Cloud Computing. *IBM Technical Report*, H-0269, 2009.

[8] C. S. Inc. Network Considerations to Optimize Virtual Desktop Deployment. `http://www.cisco.com/en/US/prod/collateral/switches/ps5718/ps4324/white_paper_c11-531553.pdf`, 2009.

[9] X. Jiang and D. Xu. VIOLIN: Virtual Internetworking on Overlay INfrastructure. In *Proc. of the 2nd Intl. Symp. on Parallel and Distributed Processing and Applications*, pages 937–946, 2003.

[10] R. Jones, K. Choy, and D. Shield. Netperf. HP Information Networks Division, Networking Performance Team, http://www.netperf.org, 2001.

[11] J. Kiszka. Towards Linux as a Real-Time Hypervisor. In *Eleventh Real-Time Linux Workshop*, Sept. 2009.

[12] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[13] M. Krasnyansky. Universal TUN/TAP device driver. `http://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/Documentation/networking/tuntap.txt`, 1999.

[14] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. D. Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12, New York, NY, USA, 2009. ACM.

[15] R. Morimoto, M. Noel, O. Droubi, R. Mistry, and C. Amaris. *Windows Server 2008 Unleashed*. Sams Publishing, 2008.

[16] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[17] R. Rose. Survey of system virtualization techniques, 2004. `http://hdl.handle.net/1957/9907`.

[18] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[19] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Virtual Distributed Environments in a Shared Infrastructure. *IEEE Computer*, 38:2005, 2005.

[20] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, 2002.

[21] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, Internet Engineering Task Force, Jan. 2001.

[22] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[23] A. I. Sundararaj, A. Gupta, and P. A. Dinda. Dynamic Topology Adaptation of Virtual Networks of Virtual Machines. In *In Proceedings of the Seventh Workshop on Langauges, Compilers and Run-time Support for Scalable Systems (LCR*, 2004.

[24] P. Ta-Shma, G. Laden, M. Ben-Yehuda, and M. Factor. Virtual machine time travel using continuous data protection and checkpointing. *SIGOPS Oper. Syst. Rev.*, 42(1):127–134, 2008.

[25] O. Titz. Why TCP over TCP is a bad idea, 2001. `http://sites.inka.de/sites/bigred/devel/tcp-tcp.html`.

[26] M. S. Tsirkin. vhost: a kernel-level virtio server. `https://lists.linux-foundation.org/pipermail/virtualization/2009-August/013525.html`, 2009.

[27] VMware. VMware VMotion and CPU Compatibility. `http://www.vmware.com/resources/techresources/1022`, 2008.

[28] B. Yassour, M. Ben-Yehuda, and O. Wasserman. Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines. Technical report, IBM Research, 2008.

[29] E. Zhai, G. D. Cummings, and Y. Dong. Live Migration with Pass-through Device for Linux VM. In *OLS '08: The 2007 Ottawa Linux Symposium*, pages 261–267, 2007.