Zero-copy Receive for Virtualized Network Devices

Kalman Meth, Joel Nider and Mike Rapoport

IBM Research Labs - Haifa
{meth,joeln,rapoport}@il.ibm.com

Abstract. When receiving network traffic on guest VMs (virtual machines) running on the KVM hypervisor, data is copied from hypervisor buffers to guest buffers. This copy incurs CPU and latency overhead. It seems desirable to avoid this copy, if possible, and to use guest buffers directly to receive network data. A number of challenges must be overcome in order to accomplish this zero-copy receive, and some additional overhead is incurred in setting it up. We present the technical details on a zero-copy receive design and implementation and discuss the resulting performance degradation. In our implementation, the savings in CPU from avoiding the copy is overshadowed by the extra handling required for the smaller and often underutilized buffers.

Keywords: zero-copy, virtual networking, KVM.

1 Introduction

In the KVM (Kernel-based Virtual Machine [1]) hypervisor, incoming packets from the network must pass through several objects in the Linux kernel before being delivered to the guest VM (Virtual Machine). Currently, both the hypervisor and the guest keep their own sets of buffers on the receive path. For large packets, considerable processing time is consumed to copy data from hypervisor buffers to guest buffers. We aim to reduce the latency and CPU overhead of delivering incoming packets by allowing the hypervisor to avoid the copy by using the guest receive buffers directly.

Zero-copy for receive (Rx) messages is not yet supported in virtio [2] – the paravirtual protocol employed by KVM. When commenting out the lines in the code that perform the copy between host buffers and guest buffers, we observed a significant drop (up to 40% for 1500-byte packets) in CPU usage. This leads us to believe that a zero-copy receive implementation could provide significant savings in CPU utilization.

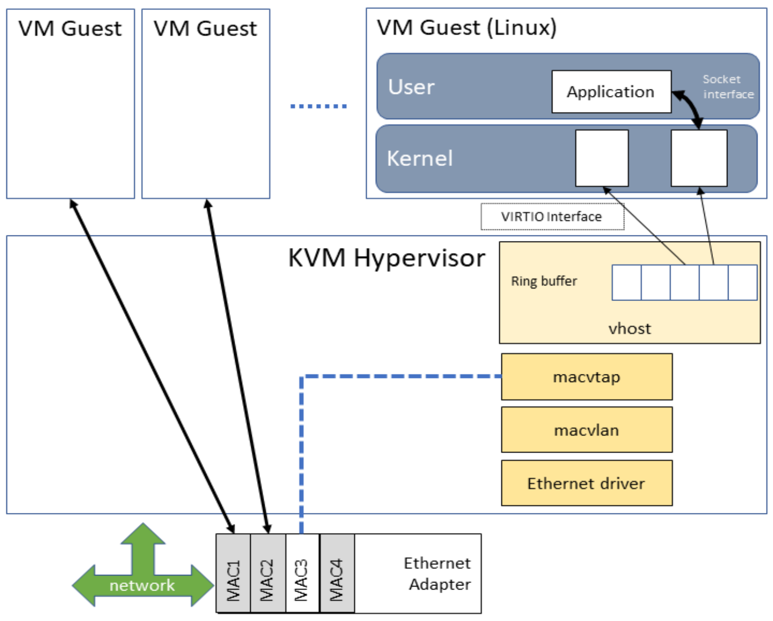

Fig. 1. High level architecture of the virtual I/O subsystem in KVM, and how it interacts with the guest OS through the Virtio protocol.

Some Linux network drivers support zero-copy for virtio on transmit (Tx) messages. Zero-copy Tx avoids the copy of data between VM guest buffers and host buffers, thus improving Tx latency. Buffers in a VM guest kernel for a virtualized NIC are passed through the host device drivers and accessed directly by the network adapter, without an intermediate copy into host memory buffers. Since the Tx data from the VM guest is always in-hand, it is quite straightforward to map the buffer for DMA and to pass the data down the stack to the network adapter driver.

A number of obstacles must be overcome in order to support zero-copy for Rx. Buffers must be prepared to receive data arriving from the network. As shown in **Fig. 1**, DMA buffers are currently allocated by the low-level network adapter driver. The data is then passed up the stack to be consumed. When data arrives, it is not necessarily clear a-priori for whom the data is designated. The data may eventually be copied to VM guest buffers. The challenge is to allow the use of VM guest buffers by the physical NIC, at least when we know that the VM guest is the sole consumer of a particular stream of data.

Recent Ethernet adapters [3] provide multiple receive queues, which may be designated for particular uses such as SR-IOV (Single-root input/output virtualization) and virtual functions [4]. We exploit this feature to map a receive queue on the adapter to a particular MAC address associated with a virtual network interface (of type macvtap [5]) of a VM guest.

Our proposed solution requires the modification of several interfaces that enable us to communicate between the high and low-level drivers to pass buffers down and up the stack, as needed. We define a VM guest with a macvtap device and assign to it a specific MAC address.

We implement a working zero-copy solution for incoming packets by enabling the network adapter to receive data directly into buffers allocated by the guest. After evaluation, we did not see significant changes in performance to warrant the effort of upstreaming these proposed changes to KVM. We identified the major sources of additional overhead that we introduced that countered any improvements gained by avoiding the copy. The major sources of additional overhead are the underutilized buffers, and the additional buffers (and associated notifications and per-buffer handling) that are required.

In the following sections we discuss other related efforts to implement zero-copy for receive, a detailed discussion of our design, evaluation of our implementation and some conclusions that we draw as a result of our experimentation.

2 Related Work

One approach to zero-copy receive that has been tried is page-flipping [6]. Data is received into buffers allocated by the device driver, and then the page with the received data is mapped into the address space of the target guest. The overhead to perform the page mapping is significant, and essentially negates the benefit achieved by avoiding the copy [7]. This is largely due to the high cost of flushing the TLB for each page that is remapped.

Userspace networking libraries (DPDK [8], Netmap [9], Snabb [10]) avoid the kernel in processing received network traffic. When used with a PMD (polling mode driver), DPDK requires exclusive access of the network device by the userspace application, which prevents sharing the adapter between applications. It also uses 100% CPU for polling, which requires a dedicated CPU.

VMWare has a VMXNET3 driver for DPDK [11]. This provides direct access to packets that arrive at the VMXNET interface (the virtual device inside the guest). Since all packets in VMWare may be routed through a virtual switch in the hypervisor (vSwitch, used for VM to VM communication), the DPDK driver cannot provide direct buffer access to the physical network adapter, because the virtual switch would be bypassed. Our tested implementation precludes the use of a virtual switch, and instead relies on the multipath feature in the Ethernet adapter to filter the packets destined only for our VM.

Tx path zero copy for virtio has been included in the upstream code almost since the beginning [12]. When it was first introduced, the performance benefit was measured to be negligible (and in some cases a regression). Socket zero-copy [13] implements zero-copy transmit semantics for the sockets API in Linux.

AF_XDP [14] (formerly Af_packet [15]) is a patchset proposed by Intel that gives a userspace program (not in a virtualized environment) direct access to the Rx/Tx rings on a hardware channel. This is relevant for multiqueue Ethernet adapters (like our solution), as each userspace program has exclusive access to the Rx/Tx rings. It does so by directly mapping userspace pages for DMA access by the Ethernet adapter (which may be a security concern). As with our solution, AF_XDP works at the L2 layer, bypassing the in-kernel network stack. This is currently not being used with virtual machines, but the potential exists to build a complementary solution since a virtual machine appears as a normal user-space process to the kernel (QEMU). There would still be a need for additional changes to update virtio rings and receive notifications.

A generic buffer allocation mechanism has also been proposed [16] [17]. This work helps define the interface that drivers will use when allocating new buffers for incoming packets. Specifically, it defines a new interface for drivers to get buffers that have been allocated external to the driver. Once existing Ethernet adapter drivers have been modified to use the new interface, our solution could benefit by using the same interface for guest allocations.

3 Design

3.1 Goals

Since our team works in an industrial research lab, we have the pressure of producing results that are immediately applicable to real deployments, or that can at least be extended to work in a real-world deployment. Our design goals thus reflect this influence, which may differ from an academic or pure research perspective.

Non-intrusive Modifications

One of the goals of the effort is to create a set of changes that could be accepted by the Linux open-source community, and integrated into the mainline codebase. We therefore aim to build a solution to work within the existing network stack framework while minimizing the impact of changes on existing code. In particular, we aim to not make any changes in the guest driver, and to not modify the existing protocols. In that way, existing VM's could continue to run unmodified and benefit from the zero-copy semantics implemented in the hypervisor. Unfortunately, we were not able to attain this goal, and the reasons will be discussed later.

3.2 High Level Architecture

To perform zero-copy receive we need to make buffers allocated by the guest available to the Ethernet adapter driver. These buffers must be placed in a receive queue owned by the Ethernet adapter that receives data specifically for this guest. This means, first of all, that the Ethernet adapter must support the ability to assign a particular receive queue to a particular guest. After setting up the association between the guest and the Ethernet adapter receive queue, guest-provided buffers must be posted to the queue. As data arrives and is placed in the guest-provided buffers, pointers to the buffers are passed up the stack to be returned to the guest. Code needs to be written in each of the intermediate layers (virtio, vhost-net, macvtap, macvlan, ixgbe) to support the new functionality.

In order to implement zero-copy receive, the physical Ethernet adapter performs DMA directly into the guest buffers. The adapter expects its individual data pages to be page-aligned, so we must be sure that the guest provides properly aligned buffers. The guest virtio_net device driver implementation supports 3 buffer management schemes (small buffers, big buffers, mergeable buffers) which are chosen depending on the capabilities of the underlying hardware. We implemented a 4th buffer scheme in the guest to provide individual 4K pages (page-aligned) to be passed from the guest to the Ethernet adapter.

3.3 Preparing a Buffer

Macvtap receives a single buffer from the vhost layer and passes it to the Ethernet adapter driver after preparing it for use. To prepare the buffer, macvtap allocates an skb (socket buffer) and attaches the data buffer to it. Skb's are used throughout the kernel to pass buffers between networking components. The macvtap driver pins the buffer in physical memory to guarantee its presence for the pending DMA operation. It then creates a mapping between the vhost descriptor number and the buffer and adds it to a hash table which tracks live buffers. This is used on the return path to notify vhost which descriptor has been filled.

3.4 Receiving a Packet

When a packet is received by the Ethernet adapter, it is sent up the network stack by the interrupt handler to determine which macvtap queue is supposed to receive the packet. The packet is attached to an skb and placed on the appropriate socket queue to be later consumed by macvtap. The skb may point to multiple buffers, as in the case of TCP reassembly. The buffers are unpinned, and we return the descriptors of the buffers to the vhost driver and the amount of data in each buffer, and then release the skb. Originally, vhost would take an skb of buffers from its socket fifo and copy the data in a packed format into the next buffers that are available in the virtio ring. We avoid the copy by having the Ethernet adapter receive data directly into the guest-provided buffers. This comes with a cost - we are not able to use the buffers in a packed format, which wastes space in each buffer. In the best case, a full 1500-byte Ethernet frame requires the use of a 4096-byte page, leaving almost ⅔ of the buffer empty.
Since we do not perform the host-to-guest copy, we have to deal with the possibility of data beginning at a non-zero offset in the buffer. This occurs when multiple buffers are returned after using TCP segmentation/reassembly offloading (TSO), in which the Ethernet headers of subsequent packets are stripped by the Ethernet adapter and combined into a large TCP packet. We therefore have to add an offset parameter to the virtio protocol to specify the offset within the page where the data starts. This wastes more space in the buffer, and requires an extension to the virtio protocol.

3.5 Push vs Pull Model

There are two possible methods for providing guest allocated buffers to the Ethernet adapter driver. When the adapter needs additional buffers (or at some time before the free buffer queue is empty), the driver may request them from the guest (a pull model). Another possibility is that the guest tries to add buffers to the free buffer queue without being prompted to do so (a push model). We implement the push model largely due to its simplicity. The pull model requires a signaling channel to be opened through several kernel objects, which can increase latency and CPU usage when requesting buffers. In addition, the pull model requires all the buffers to be pinned in advance and marked with the context of the source device. This is because the pages are pulled from the software interrupt context of the host, where we don't have the guest context available required to pin the pages.

An alternative that would influence our decision would be the addition of a pool of pre-allocated buffers. If a generic pool is implemented (as mentioned in the related work) moving to a pull model would likely be more efficient, since the pages are processed as they enter the pool.

3.6 Hardware Offloading

As is noted in LWN, in some cases a zero-copy (transmit) operation is not possible due to hardware limitations. If the Ethernet adapter is not able to perform operations such as generating checksums (in the case of TCP) or encryption (in the case of IPSec), the kernel must perform these operations instead. Any time the kernel must transform the data, it does not make sense to do a zero-copy transmission. In such cases where the kernel is already processing all of the data of a packet, copying the data is a relatively small additional cost, and zero-copy does not save any work or time. In the receive path, we have a similar dependency on hardware features. If the Ethernet adapter cannot validate the checksum of incoming packets, that responsibility falls on the kernel, which means the kernel code must be on the datapath, performing per-byte processing. This holds true for other commonly offloaded tasks as well, such as TCP segment reassembly (TSO).

3.7 Multiqueue

Our design builds upon the assumption that the Ethernet adapter has multiple queues and that we can map a particular MAC address to a specific receive queue. Most modern Ethernet adapters have multiple queues and enable dedication of individual queues to support virtual functions for VMs and SRIOV (Single Root I/O Virtualization). We exploit this feature of new adapters to map a guest VM's macvtap MAC address to a particular queue, and post the buffers from that VM to the particular queue. On Ethernet adapters that support this operation, it would be possible to perform a command like:

```
  ethtool -U eth12 flow-type ether dst 50:e5:49:3d:90:5b m
ff:ff:ff:ff:ff:ff action 3
```

This command specifies that Ethernet packets with the specified destination MAC address should be directed into the queue specified by 'action'. It turned out that the adapters we had on hand (Intel 82599) did not support this level of granularity of filtering with the ethtool command. We work around this issue by applying an existing patch [18] that enables this feature for macvtap/macvlan devices and enabling the L2-fwd-offload feature via ethtool, which allocates a dedicated queue for our macvtap device (based on MAC address). At this point, the Ethernet adapter is aware of the MAC address to be used by the receive queue for the assigned macvtap device.

3.8 Challenges

Despite the seemingly straightforward idea, there are numerous difficulties that must be overcome. When a buffer is provided by a guest, it must be mapped to its proper Ethernet ring buffer. After data is placed in the buffer, the buffer must be directed back to the proper socket interface so that it arrives at the guest that originally allocated the page. Sometimes a message arrives to a different queue, but must be provided to the guest (as in a broadcast request). We must have a mechanism to copy such a message into a guest-provided buffer. A mechanism is needed to recycle pages when a packet is corrupted and the page must be returned to its original owner for re-use. If we enable TCP segmentation and reassembly offloading, Ethernet headers of packets after the first one are bypassed by the skb (kernel socket buffer structure), but they still take up space at the beginning of the receive buffers; offsets within the buffers must now be taken into consideration when returning those buffers to the guest.

Buffer Starvation
We must be able to consistently provide sufficient buffers from user space (guest) to the Ethernet driver (in hypervisor) so that we don't need to drop packets. In the original implementation, the buffers are allocated by the Ethernet driver according to the needs of the hardware (as many as are needed, when they are needed), so this is not an issue. In our design, we rely on the guest to provide the buffers, so we must have a tighter interaction between the two modules to ensure the buffers are generated in time.

Timing
There are 3 relevant threads that interact to perform I/O on a virtio device:

- The guest virtio thread, which provides buffers to vhost and cleans up buffers that have been used by vhost
- The software (backend) interrupt thread taking the next buffer received by the Ethernet adapter and figuring out to which upper level driver the buffer should be delivered
- The vhost worker thread, which is awakened every so often by the software interrupt thread to process buffers that have been placed on its socket fifo, and also awakened by the guest virtio thread when new empty buffers are made available

We must synchronize the consumer and producer rates of each of the queues that the buffers pass through. Failure to do so results in dropped packets at one of the queues that either overflows or starves. In the original design, the Ethernet driver produces all buffers, and they are copied in a packed format to vhost buffers as they are filled, the device never starves, and fewer vhost descriptors are consumed. In our design, we have introduced a dependency on the guest to allocate buffers in a timely manner, which introduces an opportunity for starvation in the NIC driver if not fulfilled. Additionally, if the vhost thread does not always run soon after socket buffers (skbs) are filled by the NIC driver, then there is an opportunity for socket overflow. This last risk indeed occurs in our implementation as will be discussed further.

VM-to-VM Communication

Our solution relies heavily on features found in the Ethernet adapter such as switching and filtering of packets. We do so in order to take exclusive control of the Rx and Tx queues of a particular channel in a multiqueue NIC. That gives us the advantage of knowing all packets received on our channel are guaranteed to be for our VM. It also has the disadvantage of complicating VM-to-VM communication on the same host. In the case that two VMs on the same host wish to communicate, the packet flow will rely on switching all packets through the Ethernet adapter's on-board switch. This increases the stress on the hardware as traffic that is to be kept local is now counted as external traffic twice (both egress and ingress). This is the same drawback in any solution using a macvtap interface, which is identified with the MAC address of a virtual device.

4 Implementation

4.1 Virtio Descriptors

Each virtio/vhost provided buffer is identified by a descriptor. A single descriptor may be associated with multiple buffers. In the existing vhost implementation (mergeable buffers, big buffers), data is copied from host buffers to guest buffers, filling in one descriptor at time. The next descriptor to be used is specified by the vhost layer, and its data buffers are passed to the macvtap handler in a scatter-gather list. The macvtap code then copies as much data as it can that arrives on its socket to the vhost-provided buffers, returning the number of bytes written. The vhost layer then declares the descriptor as consumed and reports to the guest how many bytes were used in that descriptor. In zero-copy receive, each descriptor is associated with a single 4K buffer. The vhost layer no longer knows in advance which descriptor will be filled next, since the buffers can arrive from the Ethernet layer in any order. When a buffer is filled, we need to specify which descriptor has been consumed, and we need to specify how many bytes have been written to that buffer/descriptor. In some cases, we also have to report an offset within the buffer where the data starts (as when doing TCP segmentation and reassembly).

Some special handling in the network stack may cause buffers to be delivered out of order. Also, sometimes we need to grab a buffer to copy an skb header that has no buffers attached to it as when receiving an ARP or other broadcast packet. Vhost maintains tables of descriptors that are available and descriptors that have been used, etc. Typically, vhost reports the descriptor number and the number of bytes consumed from the specified descriptor. We added a new field to hold the offset in the buffer from which valid data begins.

The default virtio/vhost implementation provided 256 descriptors per queue, while the Ethernet adapter we used managed queues of size 512 buffers. With buffers now being provided only by the VM guest (and not by the Ethernet adapter) we quickly ran out of available buffers for DMA causing dropped packets. We increased the number of virtio/vhost descriptors to 1024 by using a more recent version of QEMU. At present, this seems to be a hard upper limit due to the implementation of virtio. Profiling our code indicates that we still sometimes run out of buffers. With increasing network bandwidth, we would benefit from the ability to allow more buffers being made available. In the traditional implementation, the Ethernet driver allocates buffers as they are needed, so the problem of running out of buffers is avoided.

4.2 Hypervisor Modifications

Each network device in Linux is abstracted by a netdev interface, which exposes the capabilities of the device, and in particular the operations that can be performed on that network object. In order to support zero-copy receive, we added 2 functions to the netdev interface. One function enables the zero copy behaviour for the particular device, and the second function enables pushing empty buffers into the device driver. These functions must be implemented for each zero-copy enabled device in the network stack that supports the netdev interface. In our implementation this includes the macvlan driver and the ixgbe Ethernet driver.

QEMU is responsible for setting up the execution environment of the VM, including any I/O devices. Emulated I/O devices are notoriously slow, so vhost was developed to bypass QEMU by providing a paravirtualized device (instead of emulated) in order to improve performance. Even when using vhost, QEMU is still used to set up the association between the frontend guest device and the backend host device. The Virtio implementation supports several types of buffer management schemes (small buffers, big buffers, mergeable buffers) depending on the capabilities of the underlying hardware. QEMU facilitates the negotiation between VM and host during VM setup to decide how to best utilize the buffers. We added an option to virtio devices to use 4K-page aligned buffers in order to support zero-copy receive. Therefore, support to negotiate this option had to also be added to QEMU.

Our implementation attempted to mimic code used for zero-copy transmit as much as possible. In zero-copy transmit, one skb is allocated in macvtap per message. Each message is associated with a single virtio descriptor. A special data structure (ubuf) attached to the skb is used to store critical information about the corresponding descriptor. The ubuf includes the virtio descriptor number and a pointer to a callback function to be called upon completion. After the data is transmitted by the Ethernet driver, when the skb is freed, the callback function is called to inform the guest that the descriptor has been consumed. In order to enable TCP reassembly by the adapter in zero-copy receive, we had to use a different scheme to recover the virtio descriptor numbers of the buffers, because multiple buffers are returned with a single skb. Instead we use a hashtable that maps each page to its virtio descriptor number. When a single skb points to multiple buffers, we look up each returned buffer in the hash table, and identify the full list of virtio descriptors that are being consumed by the buffers of the skb.

Ethernet Driver

As mentioned earlier, we modified the netdev interface by adding two new functions. These functions (ixgbe_set_zero_copy_rx and ixgbe_post_rx_buffer) were added to the ixgbe Ethernet driver.

During the setup of the macvtap device, the Ethernet driver receives a parameter containing the MAC address to which buffers will be designated. This address is used exclusively by a particular macvtap device, and has a 1:1 mapping to one of the MAC addresses on the Ethernet adapter. When the function ixgbe_set_zero_copy_rx is called, we check whether the specified MAC address matches the MAC address already assigned to one of the queues (ring buffers) of the Ethernet adapter. If it matches, we mark that ring buffer as zero-copy enabled and prepare it for zero-copy receive behaviour. We free all the buffers that were previously allocated to that ring buffer by the driver. Only buffers provided by the VM guest are subsequently posted to the ring buffer. If the provided MAC address does not match the MAC address of any ring buffer, it means that no ring buffer (queue) has been designated for that MAC address, and we return an error indication.

The ixgbe_post_rx_buffer function is called to add a guest-provided buffer to a ring buffer associated with a particular (macvtap) device. The skb parameter points to the buffer provided by the guest as well as to the device structure of the designated macvtap device. The MAC address associated with the macvtap device is compared to the allocated ring buffers. If a match is found, the buffer is mapped for DMA and is placed in the corresponding ring buffer; otherwise an error indication is returned. After the buffer is filled by the Ethernet adapter, an interrupt is generated to process the buffer and pass it up the network stack.

Macvlan/Macvtap

The macvtap driver sits on top of the macvlan driver, which in turn sits on top of the Ethernet driver. Interaction between layers is through function calls defined in the netdev interface. When macvtap is called to post a buffer, it calls the lower layer ndo_post_rx_buffer() function. This actually calls the corresponding macvlan function, which, in turn, must call the corresponding function in the Ethernet adapter driver. Similarly, for the ndo_set_zero_copy_rx() interface, macvlan simply needs to forward a reference to the macvtap device to the Ethernet adapter driver. No additional processing is required at the macvlan level, except for the patch that allows L2 forwarding offloads with macvtap [18].

In macvtap, we added two main functions: one function to send buffers down the stack to the Ethernet driver and one function to process filled buffers. In addition, we needed to make some changes to the error path to properly handle zero-copy buffers during failure. This occurs when a packet is dropped because it is badly formed or because there is no room in the socket queue for more skb's. These (now unused) buffers must also be returned to virtio with an indication that none of their space has been used.

4.3 Guest Modifications

We aimed to implement zero-copy receive with minimal changes to the surrounding ecosystem in the hope that this would ease upstreaming of our changes. In particular, we hoped to completely avoid changes to the guest VM, so that existing VMs could run unmodified. Typically, Ethernet adapter drivers provide full (4K aligned) pages to the adapter into which to place data arriving over the network via DMA (Direct Memory Access). We therefore want to take 4K aligned full pages provided by the guest and pass them down to the Ethernet driver. Under the existing implementation, there was no provision for virtio drivers to provide individual 4K aligned full pages. We therefore had to make changes to the virtio_net driver in the VM to provide full pages and to process those pages properly after they were filled with data.

In accordance with the virtio protocol, the existing virtio_net driver allocates buffers of fixed size depending on parameters negotiated with the host. The order of preference of the implementation is to use mergeable buffers, followed by big buffers, followed by small buffers. The small buffer scheme provides buffers of size about 1500, not necessarily page aligned. The big buffers scheme provides 16x4K buffers at a time for data via a scatter-gather type mechanism. The mergeable buffer scheme uses large data areas (e.g. 32K) and can pack multiple Ethernet frames of 1500 bytes into a single area. In all these schemes, data that arrive from the Ethernet adapter is passed up the hypervisor network stack to the macvtap level, and is then copied from the hypervisor buffers into the guest-provided buffers. Because the data is copied, there are no gaps in the guest-provided buffers. The code in the guest takes advantage of the fact that the data is always at the beginning of the buffer (offset = 0) and the data is consecutive. In order to implement zero-copy receive, the underlying Ethernet adapter performs DMA directly into the guest buffers. The adapter expects its individual data pages to be page-aligned, so that is what we need to provide. As discussed earlier, we implemented a 4th buffer scheme in virtio_net to provide individual 4K pages (page-aligned) to be passed from the guest to the Ethernet adapter. This results in underutilization of the buffers, and more notifications to the guest.

5 Evaluation

Our test machine is based on an Intel(R) Xeon(R) CPU E5-2660 @ 2.20GHz with 8 cores (16 hyperthreads). We allocated 8 of the threads to a single VM that ran the netperf server to ensure CPU processing power is not the limiting factor.

To evaluate our zero-copy implementation, we ran netperf TCP_STREAM over a dedicated 10Gb Ethernet connection to drive as much throughput as possible. Our baseline measurements are taken from the existing virtio/vhost implementation in Linux kernel 4.8, and we measured throughput and CPU usage by using the mergeable buffers option. This configuration gives the best performance among the existing buffer options that we described earlier.

In our first iteration of the zero-copy implementation, we observed low throughput (~4 Gbps) relative to legacy mergeable buffers (~9 Gbps). By tracing (using the ftrace tool), we observed that buffers filled by the Ethernet driver were not handled quickly enough by the vhost thread. This prevented the vhost thread from requesting more buffers from the guest, and eventually the Ethernet driver ran out of buffers. At the same time, the vhost receive queue was very full (and even overflowed at times) which lead us to realize the vhost thread was not handling the buffers as quickly as it should. We discovered that the vhost thread was running on the same CPU as the soft interrupt thread of the Ethernet driver, which was a higher priority. Therefore, the vhost thread did not get to run as long as data kept arriving on the Ethernet channel. To solve this problem, we turned off irq balancing so that the Ethernet software interrupts would likely run on the same CPU. We then assigned the vhost thread to a different CPU, in order to avoid contention. As a result, we achieved throughput with zero-copy almost on par with legacy (~9 Gbps). This points to an issue with the priorities or scheduling of the vhost thread that should have been allocated its own CPU.

We compared CPU usage for the various processes (vhost and QEMU) between zero-copy and the baseline (copy with mergeable buffers). We used 'top' to measure the CPU utilization of the threads in question. In both cases the guest (QEMU) runs close to 100% of a CPU. That includes the thread that is responsible for the virtual CPU which is running the netperf app, as well as the I/O thread responsible for the virtio ring. In the hypervisor, we see a difference in CPU utilization between the two setups. In the baseline, the vhost thread ran at about 65% of a CPU, whereas for zero-copy it ran over 90% of a CPU.

On reflecting why we might be using more CPU than the baseline, we note that we have up to 8x more vhost descriptors being used (one descriptor per 4K page instead of per 32K). We note also that we only utilize 1500 bytes out of every 4K buffer provided by the guest. The virtio_net driver in the guest (included in the QEMU measurement) must process all these buffers that are only 1/3 filled. whereas in legacy, due to the copy, buffers are fuller when processed by QEMU, and many fewer vhost descriptors are used. Despite this being less space efficient, we do not see a noticeable effect on the guest processing. However, this does not explain the higher CPU usage in the hypervisor by the vhost thread.

It should be noted that the vhost thread under zero-copy is charged with the work of posting the buffers to the Ethernet driver, while with the legacy driver, it is the responsibility of the software irq Ethernet thread. This partially explains why the vhost thead is charged with more work. But then we need to figure out how much additional CPU to charge to the mergeable case for the work done by the software interrupt handler in allocating buffers. We need to somehow measure the overall CPU usage of the entire system (all 16 cores) and then compare.

In order to measure the overall CPU usage (consumed by the guest thread, the vhost thread, and the software interrupt thread), we invert the measurement by looking at the CPU idle value of the host system. The method with the greatest CPU idle value uses the least amount of overall CPU. Here, our zero-copy implementation (87.5% idle) is quite comparable, but does not show improvement over the legacy implementation (88% idle).

6 Conclusions

We conclude that the potential savings in CPU usage from our current implementation of zero-copy is cancelled out by the extra overhead needed for QEMU/virtio to process the extra vhost descriptors (4K vs 32K buffers) and the under-utilization of the space in the 4K buffers. We believe that by addressing these inefficiencies, zero-copy can still show an advantage over the existing mergeable buffers.

As evidence towards our belief, using an MTU of 4000 to better utilize the space in the 4K buffers for zero copy helps to sometimes slightly increase our throughput (9.4Gbps for both zero-copy and mergeable buffers), but this change alone does not result in a significant improvement of CPU usage.

As the code currently stands, we don’t see a benefit to push these changes to the upstream Linux kernel.

7 Acknowledgement

This work is part of the OPERA project, which has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 688386 (OPERA) and 645402 (Mikelangelo).

References

1. KVM, "Kernel Virtual Machine," 2007. [Online]. Available: https://www.linux-kvm.org/page/Main_Page.
2. R. Russell, "Virtio: Towards a De-facto Standard for Virtual I/O Devices," SIGOPS Oper. Syst. Rev., vol. 42, pp. 95--103, 2008.
3. Intel, "Frequently Asked Questions for SR-IOV on Intel® Ethernet Server Adapters," Intel Inc., 8 March 2018. [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000005722/network-and-i-o/Ethernet-products.html.
4. C. B. Robison, "Configure SR-IOV Network Virtual Functions in Linux* KVM*," Intel Inc., June 2017. [Online]. Available: https://software.intel.com/en-us/articles/configure-sr-iov-network-virtual-functions-in-linux-kvm.
5. KernelNewbies.org, "LinuxVirt: MacVTap," KernelNewbies.org, 30 12 2017. [Online]. Available: https://virt.kernelnewbies.org/MacVTap.
6. J. A. Ronciak, J. Brandeburg and G. Venkatesan, "Page-Flip Technology for use within the Linux Networking Stack," in Ottawa Linux Symposium, Ottawa, 2004.
7. D. Miller, "David Miller's Google+ Public Blog," 18 October 2016. [Online]. Available: https://plus.google.com/+DavidMiller/posts/EUDiGoXD6Xv.
8. The Linux Foundation, "Data Plane Development Kit," Intel Inc., 2016. [Online]. Available: http://dpdk.org/.
9. L. Rizzo, "netmap: a novel framework for fast packet I/O," in Proceedings of the 2012 USENIX Annual Technical Conference, Boston, 2012.
10. M. Rottenkolber, "Snabb," Snabbco, 18 September 2013. [Online]. Available: https://github.com/snabbco/snabb.
11. The Linux Foundation, "vmxnet3-usermap," VMWare Inc., [Online]. Available: http://dpdk.org/doc/vmxnet3-usermap.
12. M. S. Tsirkin, "tun zerocopy support," Red Hat Inc., 20 July 2012. [Online]. Available: https://lwn.net/Articles/507716/.
13. W. d. Bruijn, "socket sendmsg MSG_ZEROCOPY," Google Inc., 18 June 2017. [Online]. Available: https://lwn.net/Articles/725825/.
14. B. Töpel, "Introducing AF_XDP support," Intel Inc., 31 Jan 2018. [Online]. Available: https://lwn.net/Articles/745934/.
15. A. Beaupré, "New approaches to network fast paths," LWN.net, 13 April 2017. [Online]. Available: https://lwn.net/Articles/719850/.
16. M. Gorman, "Fast noirq bulk page allocator," LKML.ORG, 4 January 2017. [Online]. Available: https://lkml.org/lkml/2017/1/4/251.
17. J. D. Brouer, "The page_pool documentation," 2016. [Online]. Available: http://prototype-kernel.readthedocs.io/en/latest/vm/page_pool/introduction.html.
18. A. Duyck, "macvlan/macvtap: Add support for L2 forwarding offloads with macvtap," Intel Inc., 17 October 2017. [Online]. Available: https://patchwork.ozlabs.org/patch/826704/.