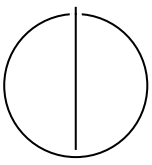
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

vDPDK: A Para-Virtualized DPDK Device Model for vMux

Dominik Kreutzer



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

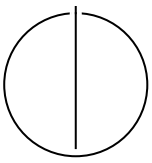
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

vDPDK: A Para-Virtualized DPDK Device Model for vMux

vDPDK: Ein paravirtualisiertes DPDK Device Modell für vMux

Author:	Dominik Kreutzer
Examiner:	Prof. Pramod Bhatotia
Supervisors:	Peter Okelmann, Masanori Misono
Submission Date:	2025-05-19



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 2025-05-19 Dominik Kreutzer

Acknowledgments

First and foremost, I want to thank my advisors, Peter Okelmann and Masanori Misono, for their invaluable support. They always encouraged my work and listened to my suggestions and ideas while also giving critical feedback, resulting in what felt like a very warm and welcoming work environment.

Second, I want to thank Prof. Pramod Bhatotia, who provided the foundations of this work environment by leading the Systems Research Group, a team of brilliant people with all kinds of fascinating research interests. I am very thankful to have found this chair and to have been given the opportunity to complete my thesis here.

Third, I want to thank my family, friends, and colleagues for their everlasting support and patience throughout my long time at TUM. Thank you, Giorgia, for being the person I can always count on to bring me joy. Thank you, Franz, for always having been both my biggest supporter and a role model to look up to. Thank you, Klaus-Peter, for your unwavering support and encouragement, helping me grow as a person.

Finally, I give thanks to the Technical University of Munich and everyone whose involvement – whether big or small – helps make it the excellent university that it is. I am grateful to the countless people who helped me learn, grow, and excel, and made me the person I am today.

Abstract

Virtual machines (VMs) are commonly used by cloud providers, which require a performant and scalable networking solution. Widely used are either device emulation, where packets are exchanged between physical and virtual NICs, or device pass-through, where a VM directly accesses a physical device. Device emulation, however, typically has a large overhead, reducing network performance, while pass-through cannot scale to a large number of VMs. In this thesis, we present vDPDK, which uses the vMux multiplexer to provide efficient, shared access of the host NIC to DPDK applications running on VMs. With vDPDK, requests by a guest DPDK application are forwarded to the DPDK backend of vMux, allowing the guest to use hardware offloads and other advanced features supported by the NIC. A fast path exists for packet transmission utilizing ring queues, and support for RX interrupts enables resource conservation and improved scalability. Our measurements show that vDPDK achieves high performance and low latencies compared to other virtual devices supported by vMux or QEMU, while scaling well to a large number of VMs.

Contents

1 Introduction

Cloud computing allows users to flexibly request and release computing resources while the cloud provider manages the underlying hardware. To give users the benefit of full operating system access as well as provide strong isolation and security guarantees, cloud providers make heavy use of virtualization. In fact, "serverless" architectures are becoming more common [1]. These systems usually consist of a large number of isolated services, requiring cloud providers to scale to even larger numbers of virtual machines (VMs) [10, 25].

A key requirement for users of cloud computing is reliable and performant network access. When virtualization is used, cloud providers must ensure this requirement is fulfilled, which raises the question of how to best provide network resources to VMs.

Typical resource management approaches when using virtualization are

- resource sharing, where a resource is concurrently used by multiple VMs,
- resource partitioning, where a resource is divided into fixed parts that are assigned to VMs,
- and resource pass-through, where a single VM is given full control of a resource.

For networking, resource sharing is typically implemented by emulating a virtual network interface controller (NIC) for each VM and forwarding the packets via a physical NIC on the VM host. This approach comes with multiple drawbacks, such as large emulation and switching overhead, as well as the inability to utilize advanced hardware features of the physical NIC.

Pass-through is a commonly used alternative, where a single VM directly uses the physical NIC on the host. While this typically performs well and allows access to all capabilities of the NIC, this approach does not scale well to large numbers of VMs, as a separate physical NIC is required for each. Resource partitioning approaches, such as SR-IOV [30] described in Section 2.1.4, partially alleviate this issue, but are still limited in their scaling [16, 27].

In this thesis, we therefore discuss the research question of whether we can performantly make use of advanced NIC capabilities in VMs while maintaining the scalability and flexibility of a resource sharing approach. In the process of this, we design, imple-

ment, and evaluate vDPDK, a new para-virtualized device usable with the vMux IO multiplexer.

This thesis is structured as follows: First, we provide required background knowledge on network virtualization and vMux. Then, we give an overview of the design goals of vDPDK and its high-level workflow. Next, we detail the design of vDPDK and how our design choices align with our goals. We then pick out particularly relevant implementation details and describe them in depth. Finally, we evaluate vDPDK, first by running experiments and comparing its performance to other network virtualization approaches, and second by summarizing related scientific work and comparing their approaches to ours.

2 Background

To provide the required background knowledge on the topic of this thesis, we describe commonly used, existing solutions for network virtualization in this chapter. Furthermore, we provide descriptions of DPDK and vMux as the core components that this thesis builds on.

2.1 Virtual machines and networking

There are various common solutions for connecting VMs to a wider network. Typically, virtual machine monitors (VMMs) can emulate a variety of NICs that the guest can use. These NICs can be based on real physical hardware or specifically designed for use in VMs. To connect the virtual NIC to the real network, packets are usually forwarded to a physical NIC. This forwarding can either be performed by the VMM or, in some cases, be offloaded to the kernel for higher performance.

Alternatively, VMs can directly use physical NICs via some kind of pass-through mechanism. Some NICs have hardware-level virtualization functionality, allowing pass-through with multiple VMs. In the following, we discuss these solutions and their differences in detail.

2.1.1 Emulation

In a typical emulation setup, the VMM will emulate a common, physical NIC. For example, QEMU [4] will by default emulate an Intel E1000. Guest operating systems (OSs) can then use the device with its regular driver. The VMM will usually forward packets between the VM and the host OS, for example, via a tap interface, and the network stack on the host will handle anything further.

This is by far the most flexible solution. By emulating a widely used device, the virtual NIC is supported by a large number of OSs. And because the virtual device is completely decoupled from any host hardware or other software, it can be used without restrictions in almost any setup. For example, this allows migrating VMs between hosts on demand [5, 14, 26].

The main disadvantage of emulating a physical NIC is poor performance [9]. Designs that work well in hardware can be difficult to performantly implement in software.

Therefore, the emulation overhead is usually high.

2.1.2 Para-virtualization

To improve on this, there are specifications for NICs that do not physically exist and are meant only for use in VMs. That means they can avoid difficult-to-emulate designs and can expect the guest driver to "know" it is running in a virtualized context. This type of device virtualization is typically called para-virtualization. The virtio-net NIC is one such device which is defined as part of the VIRTIO standard [37].

The guest OS and the virtualized virtio-net device communicate via ring queues [24]. Each ring queue is located in memory shared between the guest and host and can therefore be efficiently accessed by both. Essentially, a ring queue is a queue backed by a large array. Indices are used to keep track of the beginning and end of the queue, and they wrap around after reaching the end of the array. To communicate, one side inserts new elements at the back of the queue, and the other removes elements from the front. As long as the ring is neither full nor empty, both sides can access it asynchronously and without waiting.

Compared to emulating a real, physical NIC, this solution might be supported by fewer OSs as it requires specialized drivers. By optimizing the virtual device design and drivers for use in VMs, we can achieve much higher performance while still having the same flexibility.

2.1.3 Kernel offloading

In the previously discussed setup, every packet traverses:

1. the para-virtualized driver in the guest kernel,
2. the device emulator in the VMM,
3. the host kernel network stack.

Performance can be improved further by offloading parts of the device emulation into the host kernel.

On Linux/KVM, this can be done with the vhost-net driver. The vhost-net driver implements the virtio-net data plane in the kernel [23] and can therefore be used to bypass QEMU and directly forward packets between the guest driver and the host network stack.

2.1.4 Pass-through

Alternatively to emulation, a physical device can be passed through to the virtual machine. In that case, the guest OS can directly use a physical NIC connected to the host. When using QEMU on Linux, this pass-through can be performed with the vfio driver, which gives user-space applications like QEMU direct access to devices.

This solution usually has the highest performance [20, 9], because the host network stack is bypassed, no expensive hardware emulation is required, and the guest can use all NIC capabilities. However, this comes at a significant decrease in flexibility. The VM setup is coupled with specific host hardware, which makes migrating VMs to different hosts more challenging [40] and requires the guest OS to have drivers for all NICs that could possibly be used. Furthermore, this setup cannot easily be scaled up to more VMs because a physical NIC cannot be used by more than one VM.

To improve on the scaling of this solution, some NICs support a feature called single-root I/O virtualization (SR-IOV) [30, 33]. With SR-IOV, a PCI device can create additional *virtual functions*. These virtual functions are essentially independent PCI devices that all share the same physical hardware. In VM networking, a physical NIC can therefore be "split" into multiple virtual NICs that can be passed through to separate VMs. This way of scaling is limited, however, because the amount of virtual functions per NIC is limited [16, 27] and typically cannot be dynamically changed [40].

2.2 DPDK

The Data Plane Development Kit (DPDK) [19] is a set of libraries that can be used to directly use a NIC from user-space, bypassing the regular network stack. It can be used to send and receive packets, control device settings, and make use of hardware offloading, for example, for checksumming or flow control.

DPDK contains user-space drivers for a variety of network cards. The generic interface of DPDK allows for writing packet processing applications that are largely NIC independent. Applications can query specific hardware limits and offloading support at runtime.

2.3 vMux

vMux [28] is an application that implements NIC virtualization outside of QEMU. It can pass through host devices, but also emulate NICs for multiple VMs and multiplex their packets over a single physical NIC. Additionally, it supports mediation, which lets VMs use hardware offloads supported by the physical NIC via its emulated one.

To access the host NIC, vMux uses DPDK, bypassing the host network stack. When sending packets, vMux can therefore efficiently forward them from all guests to the physical NIC. When receiving packets, vMux uses separate RX queues for each guest and utilizes hardware offloads to sort packets into the correct queue depending on the target MAC address.

vMux can emulate both the common Intel E1000 NIC and the more modern E810 [16]. The E810 emulator also supports a mediation mode, in which some physical NIC offloads can be used by the VM. To provide the emulated device to QEMU, vMux uses the vfio-user protocol [28].

The vfio-user protocol [22] is similar to the previously mentioned vfio driver used for device pass-through. The vfio driver lets user-space applications directly access PCI devices by using a combination of ioctls, eventfds, and shared memory mappings. With the vfio-user protocol, PCI devices can be implemented in user-space. It uses the same interface as the vfio driver, but instead of accessing a real device via ioctls, requests are made to another user-space process via a socket.

Compared to regular QEMU emulation, the vMux model is still flexible and scalable, but allows for higher performance. However, there are still some drawbacks to its emulation approach.

First, mapping offloads from the virtual NIC to the physical one can be difficult or impossible. vMux needs to map specific hardware features of the emulated NICs to the more generic interface of its DPDK backend. This can be complex if the design of the hardware and the DPDK interface do not match. Furthermore, any offloads used by the guest but not supported by the host NIC need to be emulated, likely at a high cost.

Second, emulating real hardware is complex and can be inefficient. In vMux, we can observe a high emulation overhead per transmitted packet, reducing the overall performance in emulation mode.

3 Overview

Previously, we identified that common approaches to network virtualization are either not scalable (pass-through) or lack performance and hardware features (emulation). With vMux, we can use mediation to give VMs selective access to hardware offloads supported by the physical NIC. However, the emulation overhead – caused by emulating real, complex hardware – reduces performance, and it is not always clear how to map offloads of the emulated NIC to the physical NIC.

In this thesis, we research whether we can performantly make use of advanced NIC capabilities in VMs without compromising on scalability and flexibility. Towards this, we focus on providing an alternative operation mode to vMux that can be used specifically by DPDK applications to improve on the shortcomings of regular emulation. This new mode is similar to vMux's hardware emulation and mediation, but uses a para-virtualized approach based on a new virtual PCI device called vDPDK.

3.1 vMux and vDPDK architecture

To be able to give an overview of the design of vDPDK, we first describe an example vMux setup, visualized in Figure 3.1. vMux can be used with a variable number of VMs, and a virtual device is provided for each. These devices can either be one of the various supported emulated devices or a device passed through from the host. Each VMM communicates with vMux over a socket, using the vfio-user protocol.

When using an emulated device, any packet transmitted (TX) by the guest is processed by the device emulator, handed to the DPDK backend of vMux, and transmitted over the physical NIC. When packets are received (RX) by the NIC, they are steered into per-VM RX queues dependent on their destination MAC address. For performance, this is offloaded to the physical NIC. The emulated device then propagates the new packet to the VM.

When using vDPDK, the device emulated by vMux is the new vDPDK device, which we implement into vMux. Guest DPDK applications can use this device via the new vDPDK driver, which we implement into our fork of DPDK. With it, guest applications can directly access DPDK features of the host. When calling a supported DPDK function, the vDPDK driver forwards the call to vMux. Then, vMux will check the call for correctness and forward it to its DPDK backend, returning any results back

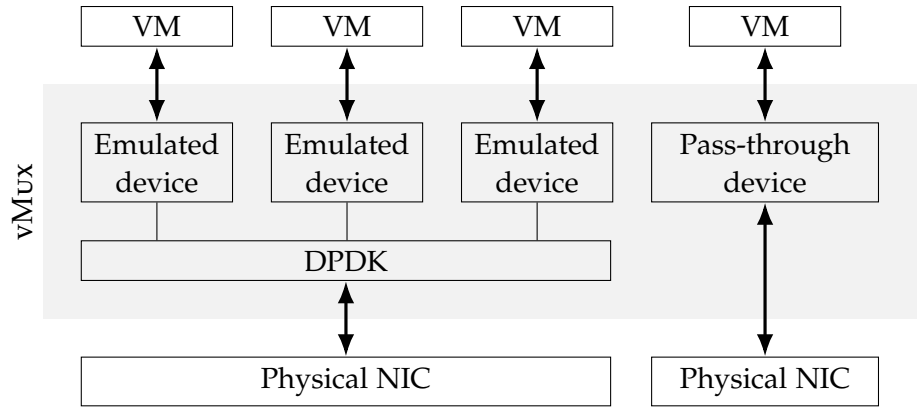


Figure 3.1: Overview of the vMux architecture.

to the guest. With this approach, guests can control various features of the NIC, such as offloads, in a device-independent fashion. We call this the *control plane* of vDPDK. Additionally, for transmitting packets, we implement a fast, asynchronous, ring-based communication protocol. Respectively, this is referred to as the *data plane* of vDPDK.

Figure 3.2 displays an overview of the architecture of vDPDK and its workflows. The control plane (blue) utilizes a *data region* – shared memory that is directly accessible by both the VM and vMux – and a *signal region* – memory that is only visible to the guest. When the guest application calls a supported DPDK function, the vDPDK driver serializes any function parameters to the data region and then accesses the signal region. This memory access causes a signal to be delivered to vMux, which then deserializes the parameters from the shared data region and – after validating the parameters – calls the same function on the DPDK backend.

The data plane (red) uses ring queues to transmit and receive packets, similar to virtio-net described in Section 2.1.2. Similarly to the control plane, the rings are located in memory shared between the guest and vMux. The data plane uses an asynchronous protocol, mainly driven by busy-polling; therefore, there is no clear order of operations. For RX, both the guest application and vMux actively poll for new packets, which is the approach favored by DPDK. Additionally, the vDPDK device polls the TX ring queue for new packets submitted for TX by the guest. For resource conservation, an event-based mode can optionally be used instead of polling in all these cases.

3.2 Design goals

We identify the following three design goals as the main challenges of this thesis:

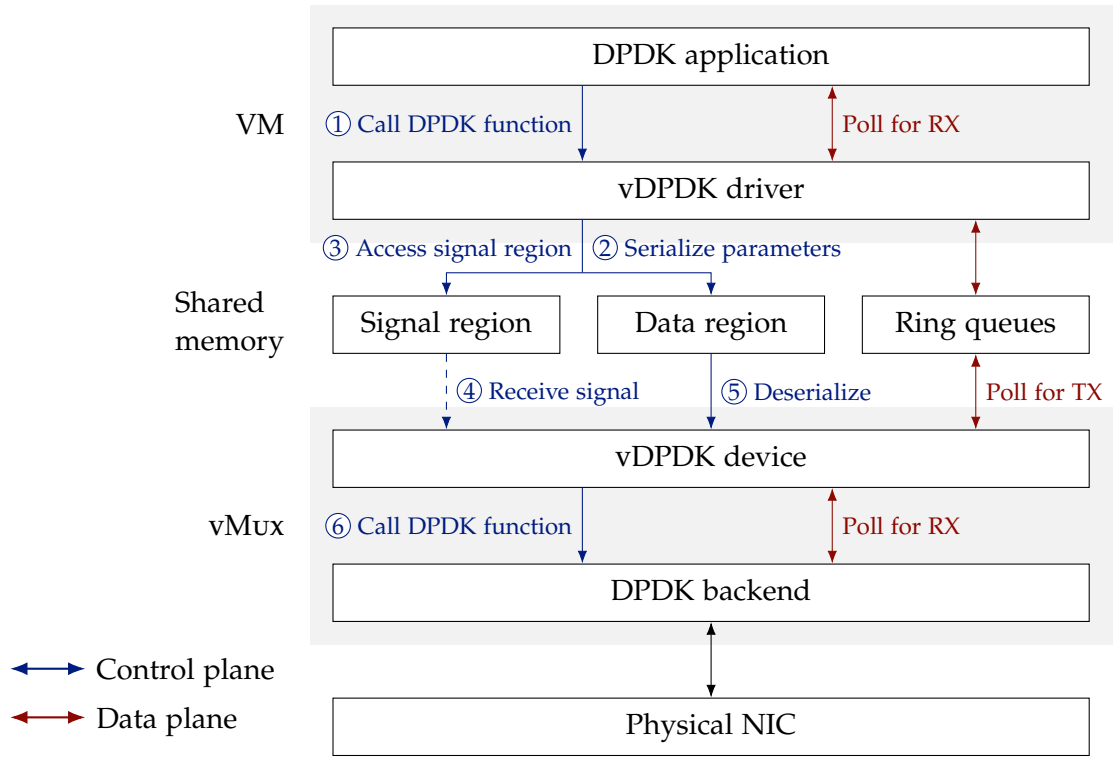


Figure 3.2: Overview of the vDPDK architecture. The control plane uses a sequential protocol to forward guest requests to the physical NIC. The data plane uses an asynchronous protocol mainly driven by concurrent polling.

Performance When applications choose to bypass the kernel network stack and utilize DPDK, it is usually because they identified high performance as a key requirement. Since vDPDK specifically targets DPDK applications, it needs to meet these performance requirements. This makes designing it challenging, as device emulation usually has a large overhead.

Our key idea is to make vDPDK performant by design. Our approach towards this is to, first, limit complexity by designing vDPDK as a thin, forwarding layer for the DPDK interface (see Section 4.2), second, use sensible data structures in the fast path for performance-critical tasks like TX and RX (see Section 4.1), and third, give applications access to offloads that can reduce the per-packet overhead.

Generality When emulating a real NIC, features of the emulated NIC cannot always be emulated with features of the physical NIC, and features of the physical NIC might not be available in the emulated one at all. When using pass-through, the VM configuration now depends on the specific hardware of the host, making migration difficult.

Our key idea is to utilize the existing DPDK API to ensure generality. Because DPDK supports a variety of different NICs, its API is already NIC-agnostic. Applications are able to dynamically discover device limits and offloads, and by using the vDPDK control plane, these can be passed through from the host NIC (see Section 4.2). This ensures that guest applications can use all features supported by the physical NIC without requiring NIC-specific code.

Scalability vMux is meant to be highly performant but also scale to a large number of VMs. Therefore, we also want vDPDK to scale well. This is a challenge because DPDK applications are usually polling-based, not event-driven.

The number of threads that can efficiently poll in parallel is limited by the number of host CPU cores. To ensure scalability, we therefore try to limit the amount of active polling threads on both the guest and host sides.

On the host side, we optimize the threading model in vMux to be able to map any number of polling tasks to a fixed number of threads, depending on the number of VMs and available CPU cores. This is further described in Section 4.3.

On the guest side, we cannot control the amount of polling as this is an implementation detail decided by the individual DPDK application. However, we implement optional support for interrupt-driven polling, which can be used by any DPDK application (see Section 4.1).

4 Design

In this chapter, we detail the design of all components of vDPDK. We describe the vDPDK data plane used for transmitting and receiving packets, the vDPDK control plane used to configure the virtual device, and our approach to scalability.

4.1 vDPDK data plane

The vDPDK data plane is responsible for moving packets between guest and host. It should be highly performant, while giving the guest the flexibility to choose different amounts of queues and queue descriptor limits. Every queue needs to allow for asynchronous communication to ensure guest and host spend as little time as possible waiting for each other. Furthermore, separate queues need to be accessible in parallel.

Following these design goals, we choose the following approach for the vDPDK data plane: Every queue is independently allocated by the vDPDK driver in guest memory that can be accessed by the virtual vDPDK device via direct memory access (DMA). The size of the queue can be chosen by the DPDK application. Each queue is structured as a ring queue that can be accessed concurrently by the guest and host without requiring a lock.

In DPDK, packets are stored in DMA-accessible buffers, called `rte_mbuf`. Typically, one packet is stored in one buffer, but it is possible to chain buffers in the fashion of a linked list if larger packets need to be sent, for example, when using a segmentation offload. In that case, each buffer in a chain is called a buffer segment. Additionally, a buffer contains fields for various metadata, mostly used for enabling and configuring offloads.

The vDPDK ring is a circular queue that holds vDPDK descriptors. Each descriptor corresponds to exactly one `rte_mbuf` segment and contains:

- IO address of the packet data, accessible via DMA
- Length of packet data
- Field for various descriptor bit-flags
- Pointer to the corresponding `rte_mbuf`

Descriptors for transmitting packets additionally contain fields that allow setting offload flags and passing offload parameters like header lengths and segment sizes.

The driver uses 16-bit indices to refer to ring slots. By design, every 16-bit number is a valid index. This means indices can be incremented and decremented freely, letting them wrap around naturally. Additionally, we ensure that the number of slots in a ring is always a power of two. This allows us to simply mask out any excess bits of the index when calculating the corresponding ring slot.

Synchronization is achieved via the descriptor flags, specifically the `AVAIL` flag. If the flag is not set, the descriptor is currently owned by the vDPDK driver on the guest. If it is set, the descriptor is owned by vMux on the host. This guarantees that there can only be one owner of a descriptor at any moment. Because only the owner writes to a descriptor, and therefore only the owner can transfer ownership, no atomic instructions or locks are required.

Frequently, both guest and host need to wait for new descriptors to be added to a ring. On the host, vMux needs to detect when new packets are enqueued to be sent by the VM. On the guest, the DPDK applications must wait for incoming packets. In vDPDK, we support two modes: polling-based and event-driven.

In polling-based mode, the ring is busy-polled in a loop to detect any new descriptors. This mode achieves the lowest latency and highest performance at the expense of higher CPU usage. It is the default mode used by DPDK applications and also used by vMux when under load.

The event-driven mode can be used to conserve resources. A DPDK application on the guest can use it by enabling RX interrupts via standard DPDK functions. In vMux, event-driven mode is automatically enabled whenever no significant load is detected on a device.

4.1.1 Transmitting packets

In vDPDK, we support one TX queue per virtual device. The vDPDK driver uses one TX ring queue, as described previously, to transfer any packets to be sent from the guest to vMux on the host. Typically, every descriptor in the ring refers to one packet. If a packet consists of multiple buffer segments, one descriptor is used for each segment. We use another flag – called the `NEXT` flag – to indicate that a descriptor is chained with the following one, forming a single packet.

Offloads are supported by copying the offload flags and metadata from the buffer into the descriptor. If supported by the host NIC, IP and layer 4 checksum calculations can be offloaded to it. Furthermore, with TCP segmentation offload (TSO), multiple TCP segments can be transferred as a single large packet. This packet will then be split into multiple checksummed packets by the physical NIC.

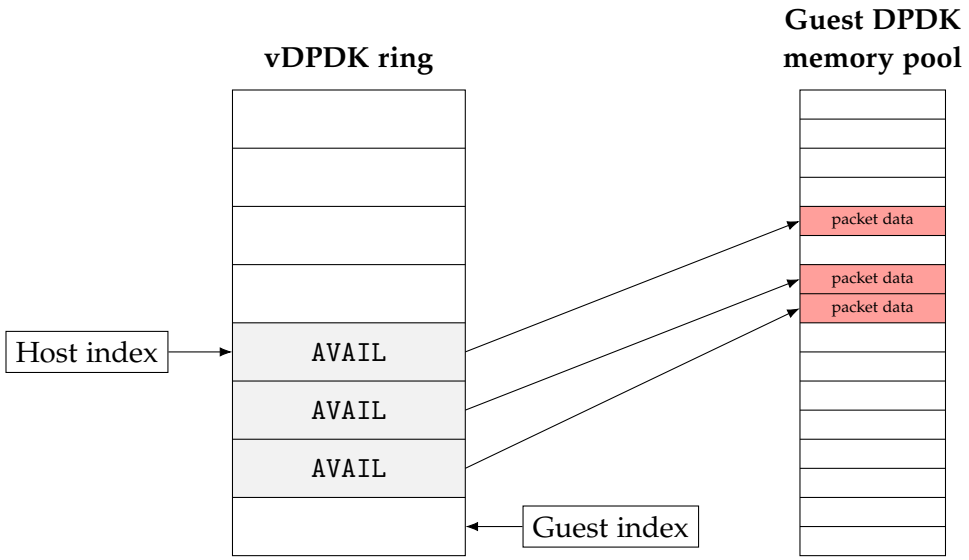


Figure 4.1: vDPDK TX ring structure.

In Figure 4.1, we show the TX queue ring during regular packet transmission. In this example, three packets were enqueued for transmission by the vDPDK driver on the guest. Each packet is contained in an allocated DPDK buffer (in red on the right). Three descriptors in the ring were initialized with the IO addresses of these buffers, and the `AVAIL` flag was set to transfer ownership to the host. The other descriptors are owned by the guest and ready to be used for transmitting packets. The guest maintains a descriptor index that indicates the next slot to be used.

On the host side, a similar index is used to indicate which slot will be checked for new packets next. Additionally, the host has its own pool of buffers, which is not depicted here. In this example, the host will transmit the three available packets during its next poll of the TX ring by copying the packet data from the guest buffers into host buffers and handing them to the host DPDK driver, which likely uses another ring queue internally.

Figure 4.2 shows the sequence of steps performed to transmit a single packet. Initially, the DPDK application on the guest allocates a buffer and fills it with packet data. It then uses a DPDK function to transmit this packet. This is handled by the vDPDK driver. The driver refers to its index to find the next descriptor in the ring to use. If it is owned by the guest, i. e. the `AVAIL` flag is not set, the descriptor is initialized with packet metadata and the IO address of the packet data. Otherwise, the ring is full and the packet is not sent. Once the descriptor is initialized, the driver sets the `AVAIL` flag to transfer ownership of the descriptor to the host and increments its index.

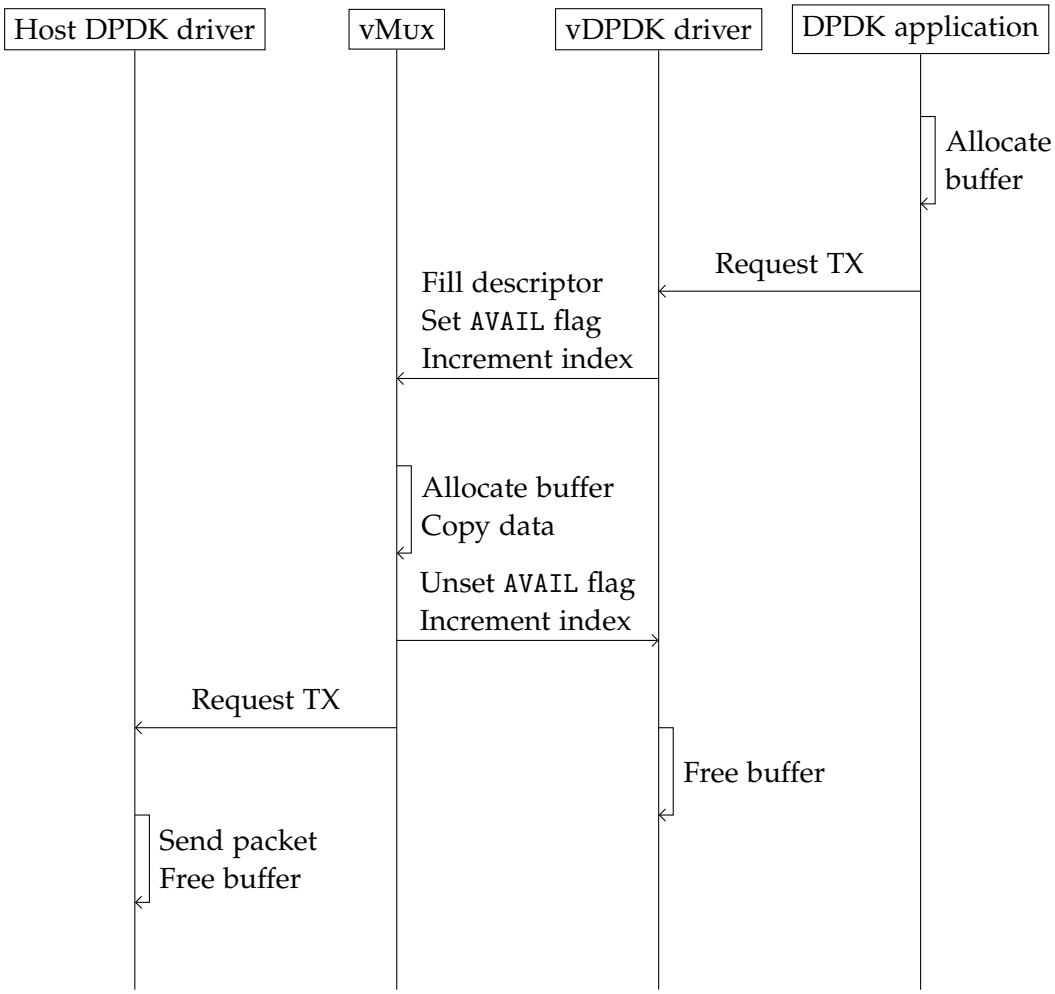


Figure 4.2: Simplified sequence diagram of the vDPDK TX protocol.

Once vMux detects that the AVAIL flag is set on the descriptor indicated by its index, it allocates a host DPDK buffer and copies packet data and metadata to it. From that moment on, the descriptor is no longer required, and therefore vMux unsets the AVAIL flag and increments its index. Then, vMux calls into its DPDK backend to instruct the driver to send the packet. The driver is then responsible for sending the packet using the physical NIC and freeing the buffer afterwards. On the guest, the vDPDK driver will eventually detect that the AVAIL flag of the descriptor is unset and free the associated guest DPDK buffer.

The exact moment when buffers should be freed is not consistently defined in DPDK, and driver implementations vary. The vDPDK driver frees buffers on two occasions: First, if the next descriptor while sending a packet still has an associated buffer, it is freed before the new buffer is attached. Second, when the amount of not-yet-freed buffers exceeds a configurable constant, all remaining buffers are freed.

In event-driven mode, a signal is sent to vMux whenever a packet is added to the ring by the driver. This feature utilizes the data and signal regions further described in Section 4.2. A flag in the data region is used by vMux to control this feature. If the flag is set, the vDPDK driver will write to the signal region whenever it adds descriptors to the ring.

4.1.2 Receiving packets

For receiving packets, the vDPDK device supports multiple RX queues. Every vDPDK RX queue is mapped to an RX queue of the physical NIC. This allows guest DPDK applications to utilize queue steering offloads that steer packets to a specific queue determined by packet information. The vDPDK driver allocates one ring for each RX queue. The ring is used to first transfer uninitialized packet buffers to vMux, and then return filled buffers for every received packet.

In Figure 4.3, we show an RX queue ring during regular packet reception. In this example, the guest driver previously allocated three buffers (in red on the right) and made them available to the host using the three linked descriptors in the ring. Two indices are maintained by the guest. The back index indicates which descriptor to check next for newly received packets. The front index keeps track of the descriptor slot that is used to pass the next allocated buffer to the host. The AVAIL flag indicates which descriptors are owned by the guest, as usual. In this example, the descriptor indicated by the back index does not have the AVAIL flag set because a new packet is available that was not polled by the guest application yet.

On the host side, only one index is used. The buffer pointed to by the descriptor indicated by this index is the one that will be filled with the next received packet. The host uses a separate set of DPDK buffers to receive packets, which are not depicted here,

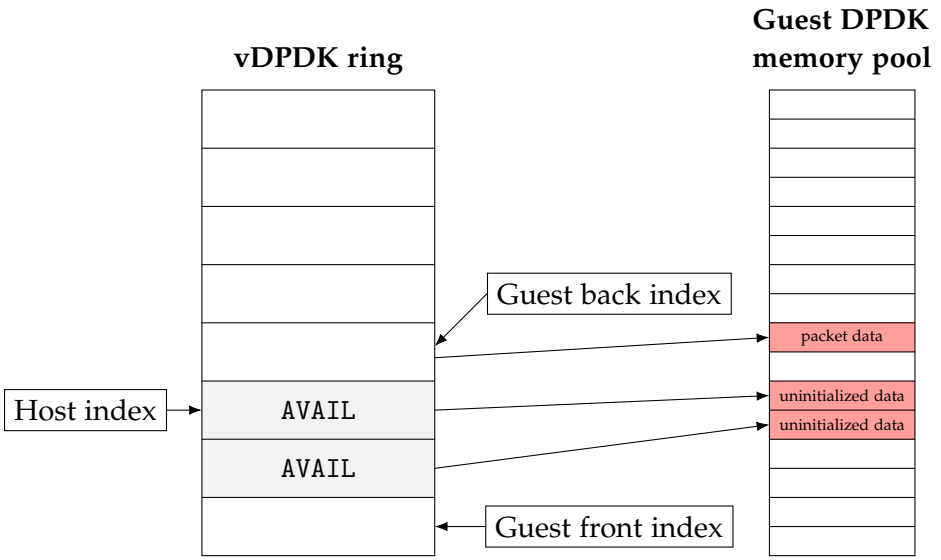


Figure 4.3: vDPDK RX ring structure.

and then copies the packets into guest buffers. In this example, the host previously received a packet, copied it to a guest buffer, and incremented its index.

Figure 4.4 shows the steps involved in receiving a packet. Note that the steps are sequentialized around the lifetime of a packet and its guest buffer, and do not necessarily occur in the depicted order. We assume a packet was previously received into a host buffer allocated by the host DPDK driver.

As the first step, the vDPDK driver allocates a buffer on the guest for the to-be-received packet. The descriptor identified by the front index is initialized with the IO address and capacity of the buffer, and the `AVAIL` flag is set. Then the front index is incremented. A free descriptor is always available for this, because in practice, these initial steps are always triggered either during the initial queue setup – in which case we can guarantee to only allocate buffers up to the ring size – or after a packet was successfully received, which frees one descriptor.

When vMux next polls the NIC's RX queue, the host driver returns the buffer with the previously received packet. Then, vMux checks if the `AVAIL` flag of the descriptor indicated by its index is set. If not, the packet is dropped. In our case, a descriptor was previously made available. The packet data is copied into the guest buffer via the IO address stored in the descriptor, and the packet length is stored in the descriptor. The `AVAIL` flag is unset to return ownership of the descriptor to the guest, and the index is incremented. The host buffer can then be freed.

The next time the guest DPDK application polls for new packets, the vDPDK driver

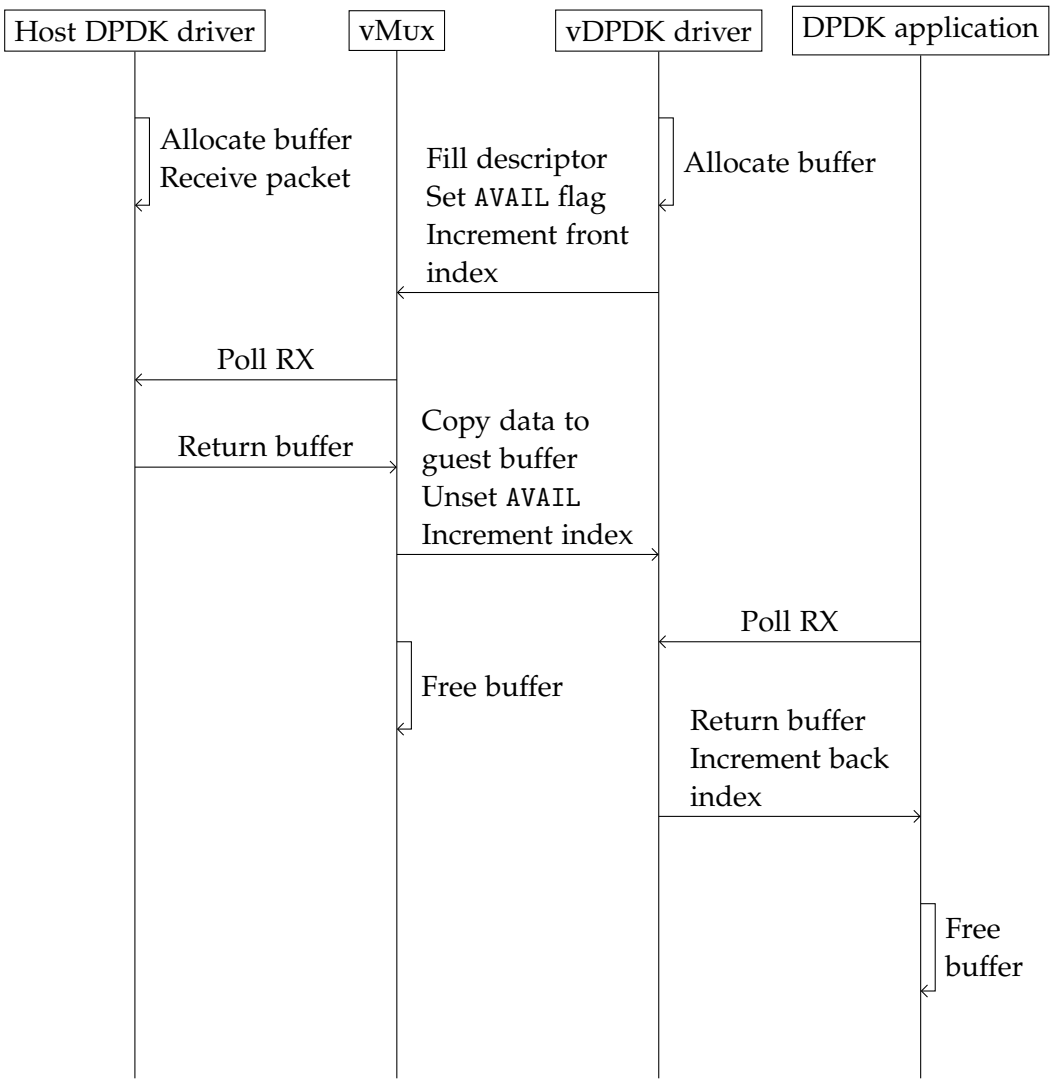


Figure 4.4: Simplified sequence diagram of the vDPDK RX protocol.

detects that the AVAIL flag on the descriptor identified by the back index is not set and is therefore owned by the guest and contains a new packet. The buffer associated with the descriptor is returned to the application, and the back index is incremented. In practice, the vDPDK driver starts over with step one at this point and allocates another buffer. The guest application is responsible for freeing the returned buffer after processing the packet.

For event-driven mode, interrupts can be delivered to the application by vMux. This feature utilizes the data and signal regions further described in Section 4.2. To request interrupts, the driver sets a per-queue flag in the data region. The exact address of the flag is determined by the queue index, which allows for independently managing interrupts for each queue. When interrupts are enabled on a queue, the vDPDK device will trigger a queue-specific interrupt after transferring new packets to the guest.

4.1.3 Zero-copy mode

As an experimental addition, we support a zero-copy mode for TX in vDPDK. In this mode, the host side of the regular TX protocol is modified to re-use guest buffers instead of copying packet data to host buffers. However, it is considered experimental and not enabled by default due to various disadvantages and unsolved design problems.

As mentioned previously, all rings and packet buffers are allocated in guest memory that is accessible by the virtual device via DMA. In vMux, all DMA-accessible memory regions of a VM are directly mapped into the virtual address space of vMux. Therefore, to access guest memory via DMA, any IO address supplied by the guest is first translated to its corresponding virtual address, which can then be accessed regularly.

The key idea of zero-copy mode is to directly use this DMA-mapped guest memory instead of copying data from guest to host. Unfortunately, we cannot directly pass guest DPDK buffers to the host DPDK backend as they contain pointers that are only valid on the guest. Therefore, a host buffer is still allocated for each packet. Any metadata, like offload flags, is copied to the host buffer. The packet data, however, is not copied. Instead, the host buffer is *attached* to the guest data, which sets the data pointer of the host buffer to the translated IO address of the packet data. And instead of immediately returning ownership of the descriptor to the guest, we delay returning it until the associated buffer is no longer used by the host DPDK driver.

The overall changes to the ring and buffer management are visualized in Figure 4.5. In this example, three packets were enqueued for transmission by the guest while another two packets were previously processed by vMux on the host. The corresponding five descriptors in the ring all have their AVAIL flag set. Therefore, they are owned by the host, and the guest vDPDK driver does not free any of the corresponding buffers.

On the host side, allocated DPDK buffers are attached to the packet data on the

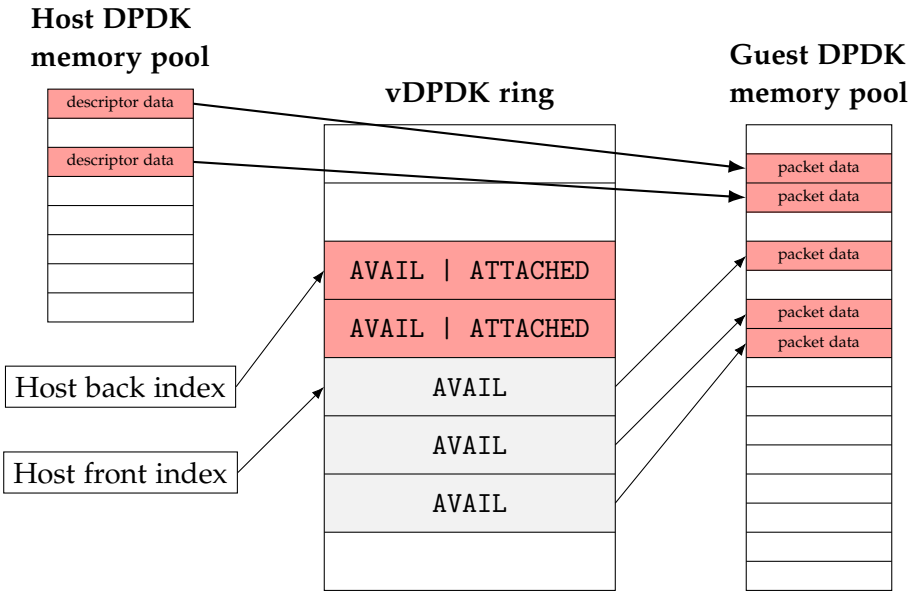


Figure 4.5: vDPDK zero-copy TX ring structure and buffer management.

guest instead of containing the data inline. To ensure that guest buffers are not freed before the packet is sent by the host driver, vMux does not unset the `AVAIL` flag on the descriptor after processing it. To differentiate descriptors that were already processed, an additional `ATTACHED` flag is used. Two indices are used: the back index indicates the next attached descriptor, and the front index shows which slot next to check for new packets.

Figure 4.6 shows the sequence of steps performed to transmit a single packet in zero-copy mode. Identically to regular TX, the guest application allocates a buffer, and the vDPDK driver copies its metadata and IO address into the ring. Then the `AVAIL` flag is set. Once the host detects the newly available descriptor at its front index, it allocates a host buffer. Instead of copying the packet data, only the descriptor is copied from the ring into the buffer. The IO address is translated and used to attach the packet data to the host buffer. This changes the data pointer of the buffer and registers a callback that is called when the buffer is freed. The `ATTACHED` flag is set on the descriptor, the front index is incremented, and the buffer is handed to the host driver via the DPDK backend.

After the host driver sends the packet, it eventually frees the buffer, which calls the previously registered callback. The descriptor – which was previously copied from the ring into the buffer – is now copied back into the ring at the back index. This is required because buffers can be freed in an order different from the order of the

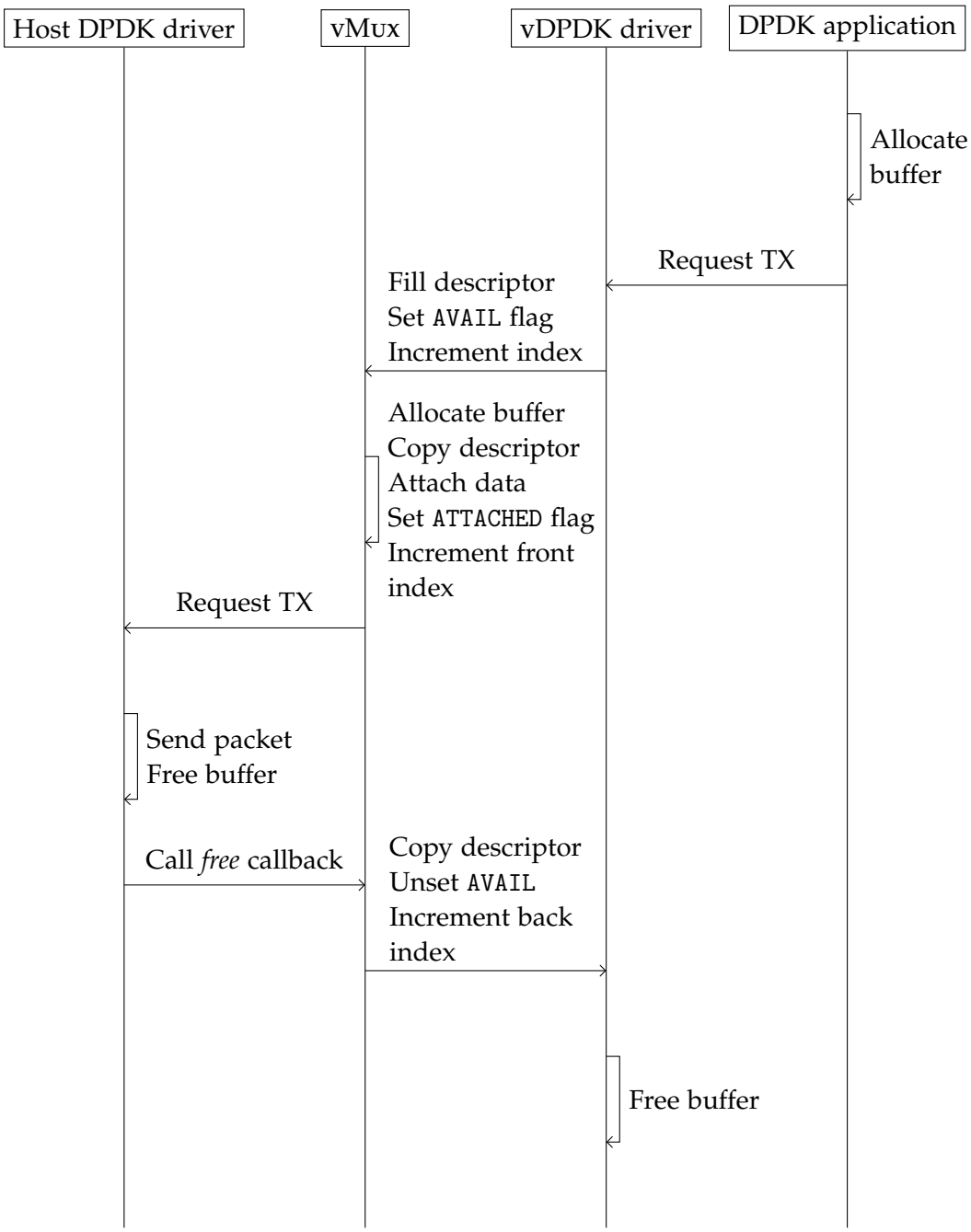


Figure 4.6: Simplified sequence diagram of the vDPDK zero-copy TX protocol.

corresponding descriptors in the ring. The `AVAIL` flag is then unset, and the back index is incremented. The guest can now free the buffer.

While this is a sound approach during regular usage, there is additional overhead compared to the default TX approach and unclear behavior in edge cases. The first issue is that, as mentioned previously, the order in which the host driver frees buffers is not defined while the vDPDK ring is a sequential structure. This means we need to be able to identify the corresponding descriptor for each buffer and reorder them if the buffers are freed out of order. We solve this by copying the descriptor into a private area of the buffer and copying it back into the ring when it is freed. This, however, means that the zero-copy mode has no advantage when sending small packets.

There is also no consistent way of managing the buffers that are still owned by the host driver. Free-threshold configurations are implemented inconsistently and are sometimes missing, depending on the driver and hardware used. There is a clean-up function, but support for it is optional, and there are no guarantees that it will free every buffer. If enough packets are sent to completely fill the vDPDK ring with attached descriptors before the driver frees any buffers, and the clean-up function is not supported, the ring is now locked forever because the driver will never free buffers until more packets are sent.

Because of this, we also cannot correctly handle changes to the DMA mappings or queue resets. DMA mappings are entirely under the control of the guest and can be changed at any time. If DMA mappings change, all data pointers in the attached buffers become invalid and need to be recalculated. This is impossible while the buffers are owned by the driver. If the queue is stopped on the guest, all attached buffers similarly become invalid.

Furthermore, if the queue is stopped and then started again, we need to ensure that descriptors from the previous queue are not inserted into the new queue ring. We handle this case by assigning each queue a unique number (nonce) and comparing it to the current queue when buffers are freed. This, however, comes with additional overhead.

4.2 vDPDK control plane

The vDPDK control plane is responsible for all device operations apart from packet TX and RX. DPDK applications perform these operations by calling the respective DPDK functions. Driver support for most operations is optional. Therefore, in our current design, we focus on supporting operations that we either expect to be commonly used or that we use in our experiments. The design is generic enough to be easily extensible in the future. We currently support operations belonging to the following groups:

- Initial device configuration and capability discovery (Section 4.2.1)
- Receive and transmit queue setup (Section 4.2.2)
- The DPDK Generic flow API `rte_flow` (Section 4.2.3)

Because we expect control plane operations to be rare and not performance critical, but have a variety of different types, a flexible, sequential, and event-based protocol is used. For this protocol, we utilize multiple BAR regions of the virtual vDPDK PCI device. The *data region* is used to transfer operation-dependent data. It is accessible as regular shared memory on both the host and guest. The *signal region* is used to signal the beginning of new operations to the host. This region is accessible as memory on the guest, but any reads or writes trigger a callback function on the host. This design allows the host to cheaply wait for new requests.

Every control plane operation approximately follows these steps: First, the DPDK application on the guest calls a supported DPDK function, which will call the corresponding implementation in the vDPDK driver. If the function has any number of parameters, they are serialized into the data region. After serializing all parameters, the guest performs a read memory access to a specific offset in the signal region. The offset identifies the type of operation to be performed. Due to the configuration of the signal region, the VMM must intercept the read, notify vMux of it, and halt the thread until the result is available.

In vMux, this triggers the signal region access callback. The requested operation is determined from the offset of the access, and any parameters are deserialized from the data region. Then, vMux calls into the host DPDK backend to serve the request. To communicate the result of the request to the guest, vMux sets the result of the original read access to the signal region. When this result is returned from the callback, the VMM completes the intercepted read, and the vDPDK driver returns control to the DPDK application.

This approach is simple and flexible, but it comes with drawbacks: First, operations can typically not be performed in parallel. Second, requests cannot be performed asynchronously. The signaling read, which starts the request, will only complete once the request is completed. Third, accesses to the signal region are slow as they need to be caught by the VMM, and delivered to vMux.

4.2.1 Initial device configuration and capability discovery

There are no control plane requests that are directly responsible for initializing or starting the vDPDK device, as the device is always considered started. Requests are made, however, to determine device information such as its MAC address and its

offload capabilities during device initialization. Offload capabilities are specified as bitflags in a single numeric value. Therefore, we can easily return them from vMux as the result of the signaling read. Similarly, MAC addresses are small enough to be directly returned. No parameters are required for these requests.

DPDK applications can dynamically discover capabilities of the used device. This includes the MAC address and supported offloads, as well as various limits such as supported buffer sizes and the maximum number of receive and transmit queues. As these device capabilities are usually statically configured per driver, and applications can therefore expect capability discovery to be fast, requests are not directly forwarded to the host via the control plane. Instead, the driver either returns values requested during device initialization, such as offload capabilities, or static values. For example, the number of transmit queues is always limited to one, and the number of receive queues is currently limited to four.

4.2.2 Receive and transmit queue setup

After initializing the device, DPDK applications set up individual receive and transmit queues. In the vDPDK driver, this allocates the ring queues in memory that is accessible by vMux and then performs a control plane request to start the queue. As parameters, the IO address and index mask of the ring, as well as the device queue index, are passed to vMux via the data region. The result of the request indicates whether the queue was started successfully.

4.2.3 Generic flow API

DPDK offers a generic flow API called `rte_flow` [7]. With `rte_flow`, applications can create rules which consist of patterns to match packets with and actions that will be performed on matching packets. Execution of these rules is offloaded to the NIC hardware. For example, this can be used to steer incoming packets to different queues depending on their Ethernet header.

To use the generic flow API, applications create rules by building chains of patterns and actions. They can then be validated by the driver before finally being applied to the NIC. This returns an opaque pointer that can then be used to remove the rule.

In vDPDK, supporting `rte_flow` requires forwarding the rules to vMux, validating and modifying them to ensure that other guests are not impacted, and then passing them to the DPDK backend. Communication occurs using regular control plane requests. The value returned by the request is a handle that the driver can use to remove the rule.

Unfortunately, serializing the `rte_flow` patterns and actions is complicated because

they are structured as linked lists containing type-erased pointers to differently-sized structs. Therefore, in this thesis, we only aim to show the feasibility of this approach by supporting the small subset of rules required in our experiments: rules that perform matching on the Ethernet header of incoming packets and steer them to a specific RX queue.

4.3 Scalable threading

When using vMux with regular emulation, two threads are spawned for every VM. The first manages the vfio-user protocol and waits on the vfio-user socket for new events. Under load, it will mostly handle packet transmission. The second thread actively polls the DPDK backend for newly received packets. In total, this means we use one polling thread per VM.

With vDPDK, we achieve high performance under load by actively polling for both packet transmission and reception instead of using an event-based approach. When using a single VM, one thread is used to poll the TX ring, and one to poll the DPDK backend for new packets. Additionally, because vDPDK is specifically targeting DPDK guest applications, we expect the VM to also actively poll for new packets. If we simply used separate threads for every VM, we would therefore expect around three busy-polling threads per VM. When scaling up the number of VMs, this would quickly exceed available resources.

We therefore design a scalable threading model for vDPDK. The core idea is to keep the number of threads constant and poll multiple VMs per thread, alternating between them in a round-robin fashion. Depending on available resources, we either poll for RX and TX in two separate threads, or alternate between RX and TX in one.

The main remaining design question is how many threads to use and how to assign VMs to threads. We dynamically decide these parameters depending on a single new vMux configuration option. For context, in regular emulation modes, vMux can be configured to pin the vfio-user and RX threads to specific CPU cores. This is used in all our experiments, in addition to similarly pinning the VM, because the host machine typically uses a NUMA architecture. A VM and its corresponding threads are always pinned within a NUMA CPU cluster. We re-use this mapping between VMs and CPU clusters to decide the vDPDK threading parameters.

As an additional configuration option, the CPU cluster for each VM can now be passed to vMux. When using the vDPDK threading model, VMs are grouped by cluster. For every cluster, RX and TX threads are spawned, and all VMs of that cluster are alternatively polled in them.

As described previously, we expect one additional polling thread in each VM using

vDPDK. We therefore estimate the amount of free CPU cores by subtracting the number of VMs from the number of CPU cores per cluster. If at least two free CPU cores are available in a cluster, two threads are spawned: one for TX and one for RX. If the number of free CPU cores is less than two, we spawn only one thread for that cluster, polling for both RX and TX.

As described previously, event-driven modes are implemented in the vDPDK device for both RX and TX. When using the vDPDK threading model, a thread waits for events only if all of the VMs polled in it enabled events. This ensures high performance but still allows for resource conservation if no load is on the system.

5 Implementation

After previously discussing the design of vDPDK, we now describe the most relevant implementation details. DPDK – and therefore the vDPDK driver – is written in C, and vMux in C++. Any source code examples from either implementation are therefore also in C or C++. We begin with details about the virtual device initialization and configuration before moving to threading and polling behavior. Finally, we describe a simple forwarding tool that is used in our experiments.

5.1 DPDK driver interface

DPDK consists of a large number of different libraries. Some provide core functionality, while others implement generic algorithms and drivers. As part of the implementation of vDPDK, we implement a driver for the vDPDK device into our fork of DPDK. This fork currently uses version 22.11 of DPDK, but we expect our driver to be easily portable to other versions of DPDK.

Similar to other DPDK network drivers, the vDPDK driver is implemented as a separate library. DPDK loads it automatically if the vDPDK PCI device is detected during start-up. Devices are identified by their PCI vendor and device IDs. Because vDPDK does not have an officially assigned ID, we use a currently unused device ID (`0x7abc`) combined with the vendor ID used by virtio devices (`0x1af4`).

To provide functionality for applications, every driver has a table mapping supported operations to function pointers. Support for most operations is optional. In most cases, each driver operation corresponds to a function of the public DPDK interface. For example, the `dev_start` operation is called by `rte_eth_dev_start`. Often, the public function is a simple wrapper, either calling the function pointer of the driver operation or returning `ENOTSUP` (not supported) if it is a null pointer. In vDPDK, we implement all functions that are required to initialize the device and transmit or receive packets. Additionally, we implement support for the control plane operations outlined in Section 4.2.

5.2 Virtual PCI device configuration

The behavior of the virtual vDPDK device is implemented in vMux. When a VM uses the device, the VMM communicates with vMux via the vfio-user protocol. This protocol is implemented in vMux with the libvfio-user library [21]. The library is responsible for sending and answering vfio-user messages over a socket. Any requests by the VMM are either passed on to vMux via a previously registered callback or answered directly according to the device configuration.

The initial configuration of the device is performed by vMux. In the following, we describe relevant parts of this configuration in detail. First, multiple BAR regions are configured. The first BAR region – called `BAR0` – corresponds to the signal region used in control plane requests. It is configured with an access callback and without any backing memory. Therefore, the VMM is required to translate any access to this region by the guest into a vfio-user message. The access callback is called whenever libvfio-user receives such a message.

The `BAR1`, `BAR2`, and `BAR3` regions form the data region used for control plane requests. They are configured without an access callback, as we expect the guest to directly access these memory regions. To allow the VMM to map this memory region into the guest, a file descriptor needs to be provided via vfio-user for each region. We use the `memfd_create` function offered on Linux to obtain a file descriptor that refers to memory without requiring a backing file or device. The file descriptor is mapped into the address space of vMux, and libvfio-user is configured to pass it to the VMM.

Currently, each data region is 4096 bytes large, which is the size of a page on our systems and therefore the smallest possible region that can be mapped. The `BAR1` region is used to pass parameters for the setup of the TX queue and contains the `TX_WANT_SIGNAL` flag, which lets vMux request that the guest driver send a signal via the signal region whenever it enqueues packets for transmission.

The `BAR2` region is similarly used for the setup of any RX queues and contains the flags that control whether incoming packets should generate interrupts on the guest. We support up to four RX queues, so four separate flags are used. Because we expect these flags to be accessed by different threads, we prevent possible false sharing problems by ensuring each flag is on its own cache line. Finally, the `BAR3` region is used to pass serialized `rte_flow` rules to vMux. This is further described in Section 4.2.3.

To be able to use MSI-X interrupts, a PCI capability with a corresponding BAR region (`BAR5`) is configured. This region is required to exist as it is referred to in the MSI-X capability configuration. However, it is never accessed via the vfio-user protocol. Therefore, we configure it with neither a callback nor a file descriptor.

5.3 Scalable threading and queue polling

To ensure vDPDK scales well to a high number of VMs, we implement a new scalable threading model into vMux. The overall design is described in Section 4.3. In summary, we use a fixed number of threads per NUMA cluster and map each VM to one. Every thread is then responsible for polling the queues of all the VMs mapped to its cluster. Depending on the number of VMs and CPU cores per cluster, either two separate threads are used for TX and RX, or one thread is used that polls for both purposes.

The vDPDK device backend of vMux implements *queue polling functions* that poll for RX and TX. These functions never block, i. e. they return immediately if no more packets are available. They also only process up to a maximum number of packets before returning. This means a thread can easily alternate polling between VMs while ensuring that individual latencies stay low. For TX, there are two separate queue polling functions: one used regularly and the other used in zero-copy mode. This allows the compiler to optimize for each case separately and ensures there is no runtime overhead for checking the current mode. At source code level, we use one function templated by a boolean parameter. This allows us to instantiate the two TX polling functions without duplicating code.

Similarly, we implement a variety of *polling loop functions* to allow the compiler to optimize for different cases. The functions differ by:

- Mode: TX only, RX only, or TX and RX combined
- VM count: one VM, or a dynamic number of VMs
- TX mode: regular, or zero-copy

These functions are mainly responsible for polling all assigned VMs in a loop until the thread is requested to stop. When starting the vDPDK threads, the correct polling loop function is chosen for each, and the thread's entry function is set to it.

Both TX and RX support an event-driven polling mode. When receiving packets, vMux can enable interrupts on the host NIC and efficiently wait until packets are received. When sending packets, the guest vDPDK driver can be instructed to send a signal (by writing to the signal region) whenever new packets are added to the TX queue. The queue polling functions dynamically enable the event-driven mode when the number of polls without new packets exceeds a configurable threshold. Since the queue polling functions are required to be non-blocking, they never wait for events directly. This is instead done by the polling loop functions.

If all queues that are polled by a polling loop function enable event-driven mode, the polling loop function stops polling until an event occurs on any of the queues. Because it is possible that RX and TX queues are polled from within the same loop, it is required

that vMux can wait for both types of events at the same time. DPDK uses epoll to wait for RX interrupts. Therefore, for TX, we use an eventfd to signal when new packets are available, as this is compatible with epoll. With the vfio-user protocol, memory accesses to certain addresses can be configured to directly trigger an eventfd with the *ioeventfd* mechanism [36], bypassing QEMU and the vfio-user socket. Unfortunately, this is not supported in our version of QEMU. Therefore, we also trigger the eventfd in the signal region access callback in vMux. To ensure that we do not miss packets that arrive in the time between the last poll and enabling events, vMux always polls an additional time after enabling events before waiting with epoll.

As described in Section 4.1, packet data is accessed via DMA. This requires mapping IO addresses to virtual addresses. However, this mapping can be changed by the guest at any moment. In vMux, DMA memory mappings are managed by libvfio-user. The library uses a callback to notify vMux of any changes. We use a lock to ensure the DMA mappings do not change while they are accessed.

No locks are used within the queue polling functions, as this would require us to frequently re-acquire and release the lock on every poll. Additionally, we must not hold the lock while waiting for an event, as this could block vfio-user communication. Therefore, this lock is best managed in the polling loop functions. We want to ensure that any locking overhead is low and that DMA mapping changes can be handled as soon as possible. To this end, we use the following locking mechanism:

A shared mutex is used as the main lock. It is either locked in shared mode by all polling loop functions or locked in exclusive mode by the libvfio-user DMA-change callback. To prevent frequent re-locking, the lock is usually kept locked by the polling loop functions, unless waiting for an event. An atomic flag is used by the callback to request access to the lock.

If the flag is set, every polling loop function releases the lock, waits for the flag to be cleared, and then re-acquires the lock in shared mode. On the other hand, when the callback is called, it sets the flag, acquires the lock in exclusive mode, and unsets the flag. After the mapping changed, it releases the lock, which lets the polling loop functions re-acquire the shared lock.

For performance, some IO addresses are not translated every time the poll function is called. Instead, they are translated once and then cached. If a polling loop function releases the DMA mapping lock, it passes a parameter to the queue polling function, which instructs it to invalidate its cache and retranslate all IO addresses. As noted in Section 4.1.3, this approach fails when using the zero-copy mode because some translated addresses are used by the host driver and cannot be invalidated.

```
struct RxQueue {
  uintptr_t ring_iova;
  uint16_t idx_mask;
  uint16_t idx;
};
std::array<
  std::atomic<std::shared_ptr<RxQueue>>,
  VDPDK_CONSTS::MAX_RX_QUEUES
> rx_queues;

struct TxQueue {
  uintptr_t ring_iova;
  uint64_t nonce;
  uint16_t idx_mask;
  uint16_t front_idx, back_idx;
};
std::atomic<std::shared_ptr<TxQueue>> tx_queue;
```

Figure 5.1: vDPDK queue management in vMux.

5.4 Queue management

In vMux, vDPDK devices are always polled by the vDPDK threads, even if no device queues were created by the guest. Only within the poll functions, it is checked whether queues actually exist. To conserve resources, the event-driven mode is considered active for any queues that do not exist.

Because the polling threads are always active and regularly access queue data, all queue management needs to be thread-safe. In the vMux implementation, queues and their state are managed using atomically replaceable, reference-counted smart pointers stored as per-device data. Figure 5.1 shows a simplified excerpt of the relevant source code.

When the guest requests the creation of a TX or RX queue, the queue data struct is allocated and the relevant smart pointer is replaced atomically. This automatically frees previous queue data, but – due to reference counting – only once it is no longer in use. When a queue is destroyed, the smart pointer is similarly atomically replaced with a null pointer.

At the start of the TX queue polling functions, the `tx_queue` pointer is atomically copied into a local variable. Reference counting ensures the pointer stays valid until the

function returns. If the loaded pointer is a null pointer, the function returns immediately because no TX queue was created yet.

When polling for RX, vMux polls the four host NIC RX queues belonging to this VM. This is always done, independent of the guest queue state. When polling a host queue and at least one packet was received, the smart pointer to the corresponding guest queue data is atomically copied into a local variable. If it is a null pointer and therefore the guest queue does not exist, we fall back to the first guest queue and copy its pointer into a local variable. This means that if packets are received on a host queue and at least one guest RX queue exists, the packet is passed to the guest. If no RX queues exist, all packets are dropped. Unless specifically configured otherwise by the guest, we expect all packets for a VM to arrive on its first host queue.

The queue data seen in the structs of Figure 5.1 is used to implement the protocols described in Section 4.1. Both queues store the IO address of the ring queue (`ring_iova`), an index mask (`idx_mask`), and various indices (`idx`, `front_idx`, `back_idx`). The IO address is used to access the ring via DMA. The index mask is derived from the number of slots in the ring queue, which is always a power of two, and is used to map indices to their corresponding ring slot by masking out the excessive bits. The indices are used as described in Section 4.1. Note that the front and back indices of the TX queue are only used in zero-copy mode. Otherwise, the front index is ignored, and the back index is used as the regular index. The TX queue additionally contains a number (`nonce`) that is guaranteed to be unique to this queue instance, i. e. it is guaranteed that two separately created TX queues of the same device can be differentiated using this number. This is used for zero-copy mode, as described in Section 4.1.3.

As seen in Figure 5.2, similar queue data is used in the vDPDK driver on the guest. No synchronization is required for queue setup as it is not considered to be a thread-safe operation in DPDK. The `private_data` member provides access to per-device data, such as the addresses of data and signal regions. The ring is allocated in DMA-accessible memory and managed using a `rte_memzone`, which not only provides a pointer to its memory, but also the associated IO address needed by vMux to access the ring. The `idx_mask` and indices are used similarly to their counterparts in vMux described above. In DPDK it is the responsibility of the driver to allocate buffers for RX, and therefore, the RX queue data contains a memory pool (`pool`) from which to allocate new buffers. The `alloc_descs` member of the TX queue data keeps track of the number of descriptors that are associated with an allocated buffer. This is used in combination with the configurable `tx_free_thresh` value to free buffers as described in Section 4.1.1.

```
struct vdpdk_rx_queue {
  struct vdpdk_private_data *private_data;
  const struct rte_memzone *ring;

  uint16_t idx_mask;
  uint16_t front_idx, back_idx;

  struct rte_mempool *pool;
};

struct vdpdk_tx_queue {
  struct vdpdk_private_data *private_data;
  const struct rte_memzone *ring;

  uint16_t idx_mask;
  uint16_t idx;

  uint32_t alloc_descs;
  uint16_t tx_free_thresh;
};
```

Figure 5.2: vDPDK driver queue data.

5.5 Lazy DMA mapping for zero-copy

A NIC typically accesses packet data via DMA. Therefore, by default, DPDK allocates its packet buffers in DMA-accessible memory. With vDPDK, a DMA memory mapping is set up via the vfio-user protocol and used to access guest packet buffers. Similarly, the host DPDK backend of vMux allocates its buffers in DMA-accessible memory, which is then accessed by the physical NIC. When using the zero-copy mode, the packet data on the guest is attached to a host DPDK memory buffer, as described in Section 4.1.3. The main challenge here is that, unlike the host packet buffers, the attached data residing in the guest DMA memory mappings is not necessarily DMA-accessible by the physical NIC.

In DPDK, any memory that is not directly managed by DPDK is called *external memory*. To use such memory, it must be registered with DPDK, which is easily achievable with the `rte_extmem_register` function. This function, however, does not ensure that the external memory is accessible with DMA. The method to achieve this depends on the kernel driver that DPDK uses to access the physical NIC. The driver that is recommended to be used on Linux is vfio, and for this case, DPDK provides functions to map external memory to a chosen IO address and thereby make it DMA-accessible. To avoid more complex memory management, we simply re-use the virtual address of the external memory as its IO address. We do not implement this for other drivers; therefore, zero-copy mode is currently only supported when using the vfio driver with an IOMMU.

The remaining challenge is identifying which memory to map for host DMA access. As a simple solution, we can map all memory that is likewise mapped by the guest and unmap it when it is unmapped by the guest. Unfortunately, it is common for the guest mappings to change frequently while the VM boots. In our initial implementation, we observed large slowdowns caused by the frequent host DMA mapping and unmapping, which in turn caused vfio-user requests to time out. Additionally, it is unclear if this method would scale well to a larger number of VMs.

As an alternative, we only map memory regions that contain packet data. Because it is impossible to determine up front where the guest will allocate packets, we implement a lazy DMA mapping method. If zero-copy mode is enabled, the vDPDK device keeps track of all DMA regions that are mapped by the guest. Whenever a packet is transmitted and its IO address is translated, the device finds the DMA region that contains the packet data. If this region was encountered for the first time, it is then DMA mapped for host devices to access. The mapping is kept until the guest removes the region.

During testing, we observe that typically only one such region is mapped when starting a new DPDK application on the guest and sending the first packet. All

following packets are usually located within the same region.

5.6 vDPDK descriptor layout

In the vDPDK data plane, we use descriptors located in ring queues to transfer packet information between guest and host. The general purpose and design of these descriptors are further described in Section 4.1. In the following, we provide further details of the choices made when implementing the vDPDK descriptors.

Descriptors are located in shared memory and accessed by both the guest and host. It is therefore required that both sides agree on the exact memory layout of a descriptor. To guarantee this in all cases, care must be taken to ensure all accesses to descriptor data are performed in a platform-independent fashion. This complicates the implementation and may incur overhead.

Alternatively, we can place restrictions on which types of guest systems are supported and ensure that their binary interface matches that of the host. It is unclear which option will be best for vMux in the long term. For our initial implementation, we try to keep the amount of restrictions placed on the guest low while keeping the code simple.

The design of the vDPDK data plane requires that descriptors contain data used by both host and guest, but can also be associated with guest DPDK buffers. We therefore split a descriptor into two parts: shared data and guest-private data. The host must treat the guest-private data as opaque, i. e. preserve it, but not modify or interpret it.

Figure 5.3 lists the descriptor definitions from the vDPDK driver. To ensure the memory layout of these structs is the same as expected by vMux, we use static asserts to validate the offsets of all struct members. Endianness is assumed to be the same between host and guest. Both TX and RX descriptors have fields in common. The `dma_addr` field contains the IO address of the packet data, and the `len` field contains its length. The `flags` field contains descriptor flags, most notably the `AVAIL` flag used for synchronization. The guest-private `buf` pointer identifies the associated guest DPDK buffer. It is a null pointer if the associated buffer was freed previously. As a special case, the `len` field of the RX descriptor is initially set to the capacity of the associated buffer. Only after a packet was received and the descriptor was returned to the guest, the field contains the length of the received packet.

The TX descriptor additionally contains fields for parameters concerning per-packet offloads. The `offload_flags` field in particular controls exactly which offloads should be enabled for a packet. The different offload flags are defined by DPDK and not translated by the vDPDK driver. We therefore require that the offload flag definitions of the host and guest DPDK versions are identical. This restriction could be lifted in the future by mapping between DPDK and vDPDK offload flags.

```
struct vdpdk_tx_desc {
  // Common descriptor data
  uint64_t dma_addr;
  uint16_t len;
  uint16_t flags;
  // Offload parameters
  uint16_t tso_segsz;
  uint8_t l2_len, l3_len;
  uint64_t offload_flags;
  uint8_t l4_len;
  // Guest-private data
  struct rte_mbuf *buf;
};
```

```
struct vdpdk_rx_desc {
  // Common descriptor data
  uint64_t dma_addr;
  uint16_t len;
  uint16_t flags;
  // Guest-private data
  struct rte_mbuf *buf;
};
```

Figure 5.3: Annotated TX and RX descriptor definition from the vDPDK driver.

Note that the additional descriptor fields are arranged to minimize padding between fields and keep the size of the descriptor low. Even so, the TX descriptor is nearly twice as large as its RX counterpart. To reduce the size, it is possible to use an alternative approach. If we require that the DPDK `rte_mbuf` definitions on guest and host are binary compatible, we only need to pass the IO address of the buffer in a descriptor. All other data can then be read directly from the header of the buffer. We leave potential implementations of this to future work.

5.7 Ring queue synchronization

The ring queue is the central data structure of the vDPDK data plane. It can be efficiently accessed and modified in parallel by both the guest and host. No typical locks or read-modify-write atomic operations are required. Instead, we use an ownership-passing model. At any moment, every descriptor in the ring has a single owner. Only the owner is allowed to modify the descriptor. And only the owner can pass ownership to another party. Because there is always only a single owner, no conflicting parallel modifications can occur. It is also impossible for two parties to become owners at the same time, because the next owner is always explicitly chosen by the previous owner.

In vDPDK, there are only two possible owners for each descriptor: the vDPDK driver on the guest and vMux on the host. The current owner is identified by a single bit in the `flags` field, shown in Figure 5.3. We call this the `AVAIL` flag. If the flag is set, the descriptor is owned by vMux. Otherwise, it is owned by the guest driver.

On the host, vMux only uses a descriptor if its `AVAIL` flag is set. After using the descriptor, vMux unsets the flag. Similarly, the guest driver only accesses descriptors where the `AVAIL` flag is unset. Only after all accesses to the descriptor are completed does the driver set the `AVAIL` flag to pass ownership to vMux.

For this to work correctly, our implementation needs to uphold two guarantees. First, setting or unsetting the `AVAIL` flag needs to be atomic, i. e. changing the flag never has any observable intermediate effects. We expect this to be the case in any sensible CPU architecture. Second, all descriptor modifications must be made while the party has ownership of the descriptor, i. e. we must ensure that neither the compiler nor the CPU moves descriptor accesses outside of this ownership window. We implement this with memory barriers.

DPDK provides functions for accessing hardware registers. These already contain appropriate memory barriers. In the vDPDK driver, we therefore ensure that all reads of the descriptor flags – if they test for ownership – are performed with the `rte_read16` function. Similarly, when passing ownership, we always use the `rte_write16` function.

5.8 vDPDK RX interrupts

DPDK provides functions to enable and disable RX interrupts on a device queue if the NIC driver supports this. This can be used by an application to either conserve resources or facilitate scalability. As described in Section 4.1, this feature is implemented by the vDPDK driver. Interrupts are controlled per queue by setting a flag in the control plane data region.

A DPDK application that enabled RX interrupts can use epoll to wait for them. Internally, – if the device is accessed using the vfio driver – the DPDK driver creates a mapping between queue numbers and interrupt vectors, and DPDK associates a file descriptor with each vector. Drivers other than vfio might require a slightly different approach. In vDPDK, we implement a simple, fixed mapping that associates interrupt vectors one to four with RX queues one to four. Note that interrupt vector zero has a special function in DPDK and cannot be used for this. This implementation is fully functional when using the recommended vfio driver. Other drivers may not work as expected.

In the vDPDK device implementation in vMux, supporting interrupts requires configuring libvfio-user correctly. Because of the fixed interrupt vector mapping, this is fairly simple. An MSI-X capability is added to the PCI configuration space, and an associated BAR region is configured. This region must exist for correctness, but it is never accessed when using the vfio-user protocol. In the RX poll function, the device records which queues received packets and – after the full batch of packets has been

passed to the guest – triggers an interrupt for each such queue by using a provided libvfio-user function.

5.9 DPDK TAP forwarding application

For this thesis, we implement an example DPDK application that can be used to test our implementation with a variety of regular networking tools. This application accesses a NIC with DPDK and creates a TAP device for it. The TAP device is usable as a regular network interface by other applications. Effectively, this can be used as a simple user-space vDPDK driver. However, the implementation is not vDPDK-specific and works with any DPDK driver supporting the required operations.

Essentially, this application is a forwarder. Any packets received on the DPDK device are written to the TAP device and, therefore, "received" by the kernel network stack. Similarly, any packets that are sent via the TAP interface are read by the forwarder and sent using the DPDK device.

While this can be achieved with existing tools like FastClick [3], our forwarder specifically makes use of the features provided by vDPDK. Most importantly, we use RX interrupts to conserve processing resources. This especially helps performance on VMs with a low amount of vCPUs. In our implementation, interrupts are enabled whenever no packets were received during the last RX poll.

The application also reads device information and applies it to the TAP interface. This automatically configures the correct MAC address on the TAP interface, which, when using vDPDK, was forwarded from vMux. Additionally, we forward information about offloading capabilities. Specifically, we support offloads for TCP and UDP segmentation, as well as checksumming offloads.

To use offloads with the TAP interface, we advertise the available offloads to the kernel with the `TUNSETOFFLOAD` ioctl. We additionally configure the interface to prepend a virtio-net header to every packet. This header contains information about which offloads are used by the packet and related parameters. On the TX path, the kernel network stack can then use any of the supported offloads. The forwarder reads the offload information from the virtio-net header and configures the DPDK packet buffer accordingly. On the RX path, we currently do not use any offloads. The virtio-net header is still required to be prepended to the packet, but it is simply zero-initialized.

Most relevant to our tests is the TCP segmentation offload. With this offload, a driver can pass a large TCP packet to the NIC, which segments the large packet into multiple regular TCP packets. In our case, it allows our application to read multiple packets of TCP data from the TAP device with a single system call, which significantly reduces per-packet overhead. Additionally, due to vDPDK forwarding offloads to the host NIC,

we also reduce overhead in the guest and host drivers.

Packet buffer handling of the forwarder is optimized to reduce the amount of data copies and allow for efficient use of offloads. Specifically, on the TX path, this means we read packets from the TAP device directly into a DPDK buffer. The virtio-net header is removed from the packet by increasing the buffer headroom, which is an offset that is used by DPDK to determine the start of the packet. To take advantage of TSO, we also use the largest possible size for the DPDK buffers. On the RX path, we prepend the virtio-net header to the packet by using the default DPDK buffer headroom.

For threading, we use the existing DPDK threading model, and adapt to the number of available threads. If at least two threads are available, TX and RX are handled on two separate threads. If only one thread is available, both paths are handled on the same thread. To ensure that reading the TAP interface does not block the thread, the associated file descriptor is set to non-blocking mode, and epoll is used to determine when a packet is available.

6 Evaluation

In Chapter 3 we describe our three design goals: performance, generality, and scalability. In this chapter, we aim to evaluate whether our design and implementation of vDPDK achieves these goals. We run automated experiments to compare vDPDK to other vMux emulation and pass-through modes, as well as other common virtualized networking solutions.

6.1 Experiment setup

All experiments are run on two physical machines: the load generator and the VM host. On both machines, an Intel E810 NIC is used for the experiments. The NICs are directly connected with a link speed of 100 Gbit/s. The VM host is responsible for running one or multiple VMs, possibly with vMux. On the VMs, various experiments are run that typically handle traffic generated by the load generator.

The VM host uses two AMD EPYC 7413 CPUs with a total of 48 cores – SMT is disabled – and 2 TB of memory. Every cluster of 6 cores shares the same L3 cache. Therefore, when running VMs, we ensure that QEMU, as well as the vMux threads communicating with it, are pinned to one cluster. The VMs are distributed across all clusters.

The experiments are controlled from a third machine. It is connected to the VM host, the load generator, and all VMs via a separate management network. Python scripts are executed on the controlling machine, which automate the entire experiment by executing commands on all involved machines via SSH.

We compare vDPDK with other vMux modes as well as other common networking virtualization approaches, an overview of which is given in Table 6.1. These approaches mainly differ by the type of virtual device observed by the VM, and the method used to access the physical NIC on the host. The virtual device is either an emulated physical NIC like the Intel E1000 or E810, a para-virtualized device like virtio-net or vDPDK, or a physical NIC passed through from the host.

All virtual devices access the network via the physical NIC on the host. When using vMux, it handles physical device access with its DPDK backend, unless it is used in pass-through mode. Alternatively, QEMU can directly use vfio to pass through a device.

Name	Virtual device	Physical device access	Notes
vMux-pt	Pass-through	vMux	
vMux-vDPDK	vDPDK	vMux	No kernel driver
vMux-emu-e1000	E1000	vMux	
vMux-emu-e810	E810	vMux	
vMux-med-e810	E810	vMux	Mediation of hardware offloads
Qemu-pt	Pass-through	vfio	
Qemu-VirtIO	virtio-net	Linux bridge	
Qemu-vhost	virtio-net	Linux bridge	Kernel offloading with vhost-net
Qemu-e1000	E1000	Linux bridge	

Table 6.1: Summary of tested virtualization approaches.

When using the virtual devices implemented by QEMU, they are connected to the physical NIC via a bridge, an in-kernel network switch offered by Linux.

6.2 DPDK benchmarks

As the vDPDK device is mainly designed for use in DPDK applications, we first evaluate its performance using experiments implemented with DPDK. We focus on two pre-existing DPDK applications: MoonGen and FastClick.

MoonGen [8] is a scriptable packet generator that uses DPDK to access the NIC. With Lua scripts, a wide variety of packet generation and processing tasks can be performed. Additionally, MoonGen supports high-precision packet timestamping, which we use to take precise latency measurements. FastClick [3] is a modular router that can also perform a variety of packet processing tasks. Packet pipelines are created using simple configuration files. To conform with the generality design goal, no vDPDK-specific changes were made to these applications besides building them with our fork of DPDK.

6.2.1 Throughput and latency

With this test, we want to evaluate the packet throughput of vDPDK. We use MoonGen on the load generator to send a fixed-rate flow of 60-byte packets to a single VM. On the VM, a reflector is used to reflect any packets received on the virtual NIC back to the load generator. This is achieved by simply swapping its source and destination MAC addresses and retransmitting the packet. Because vDPDK is designed for use

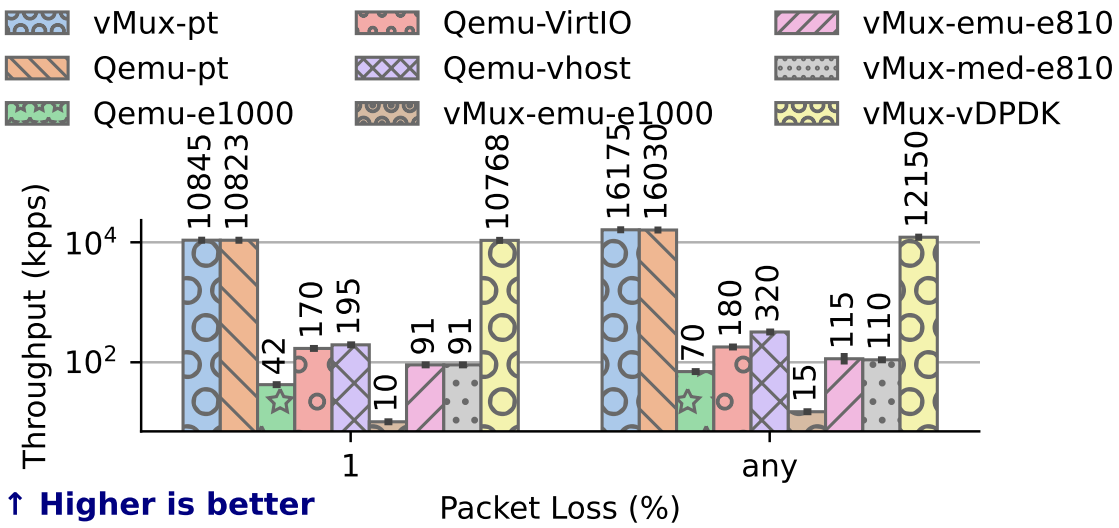


Figure 6.1: Results of throughput measurements in packets per second.

with DPDK applications, MoonGen is used as a reflector for it and other suited virtual devices. For other devices, packets are reflected in kernel-space with XDP.

On the load generator, we measure the rate of incoming and outgoing packets. This lets us determine the throughput and the rate of packet loss. Additionally, by sending timestamped packets and recording the arrival time of its reflected counterpart, we take precise latency measurements.

To estimate the rate at which we begin to observe significant packet loss, we run the experiment multiple times with increasing packet rates. During every experiment run, we take measurements for 30 seconds, and each run is repeated at least once.

Figure 6.1 shows the throughput measured during this experiment. We display an estimate of the maximum packet rate that incurs less than one percent packet loss on the left, and the overall maximum measured packet rate on the right. As expected, the two pass-through modes achieve the highest packet rate overall, peaking at 16 Mpps. QEMU can reach packet rates of up to 320 kpps when using virtio and vhost, while the vMux E810 emulator reaches 115 kpps. However, when using vDPDK, vMux can achieve packet rates comparable to pass-through modes. At one percent packet loss, we measure 10 Mpps for pass-through modes and vDPDK. Overall, vDPDK achieves only slightly less throughput at 12 Mpps.

The latency measurements, displayed in Figure 6.2, show similar results. Pass-through modes and vDPDK achieve very stable, low latencies below 10 µs, even in the worst case. Otherwise, Qemu-vhost and vMux-emu-e810 show the lowest latencies,

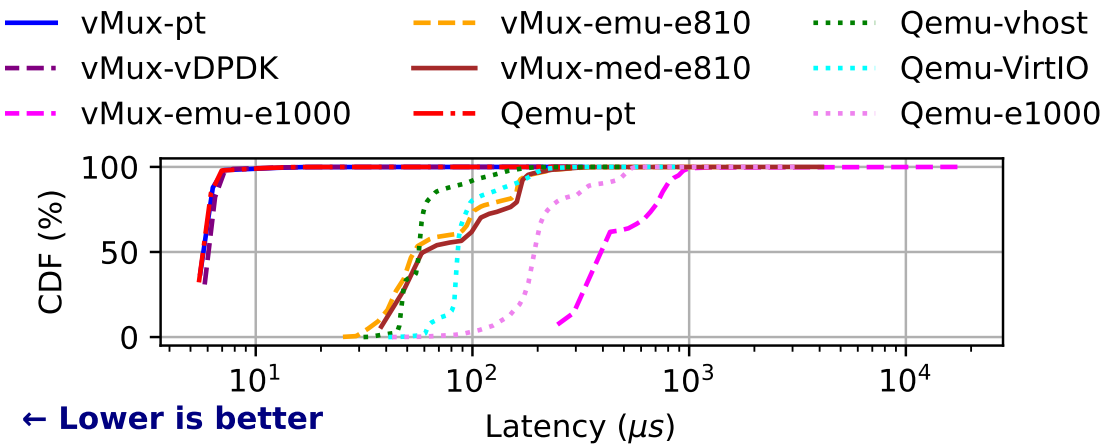


Figure 6.2: Latencies measured during the throughput experiment.

both with an average of below 100 µs.

Overall, these results show that we succeed in achieving the performance design goal. When running DPDK applications, vDPDK can achieve high packet rates with low latencies, comparable to pass-through solutions.

6.2.2 Packet classification

To measure scalability as well as the benefit of the offload pass-through offered by vDPDK, we test the performance of a simple packet classification task running on one or multiple VMs. On the load generator, MoonGen is used to generate packets with varying ethertypes at a total packet rate of 40 Mpps. The packets are sent to all VMs in a round-robin fashion.

On every VM, an instance of FastClick is used to classify packets into four groups depending on their ethertypes and count the number of received packets. Three different FastClick configurations are used:

- Hardware classification using the generic flow API (`rte_flow`) of DPDK
- Software classification with DPDK
- Software classification with kernel devices

If the virtual NIC can be accessed by FastClick with DPDK, we test both hardware classification using `rte_flow` and software classification. Otherwise, FastClick accesses the NIC via its kernel device, and only software classification is used.

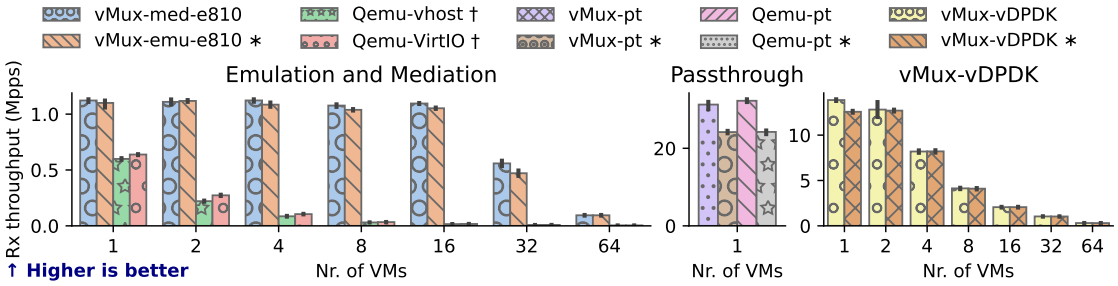


Figure 6.3: Per-VM RX throughput during packet classification using `rte_flow` (unless otherwise noted), FastClick with DPDK (∗), or FastClick with kernel devices (†).

Figure 6.3 shows the average per-VM RX throughput during classification in packets per second depending on the number of VMs and virtual device used. On the left, different emulation and mediation solutions are compared. We measure that regular vMux reaches around 1.1 Mpps with one VM, and scales well up to 16 VMs with no significant loss of performance. When using QEMU emulators, we only reach 0.6 Mpps with one VM, and performance quickly falls off as we scale up.

The next graph shows the performance of pass-through methods. Once again, they have the highest performance with packet rates of up to 32 Mpps. The benefits of hardware classification are also apparent. Classifying packets in software reduces the packet rate by 25 % to around 24 Mpps. Pass-through methods, however, cannot scale to more than one VM, which is their main drawback compared to all other solutions.

Finally, on the right, we display the performance of vDPDK. With one VM, we reach 14 Mpps with hardware and 13 Mpps with software classification. This shows that our support for `rte_flow` has a small but measurable performance benefit. It is likely that a larger benefit could be observed if vDPDK were further optimized towards higher packet rates in the future.

When scaling up the number of VMs, vDPDK stays equally performant at two VMs, and then performance decreases steadily. Note, however, that RX throughput with 16 VMs is still at 2 Mpps, faster than all other emulation and mediation modes. Furthermore, with any number of VMs, vDPDK performs better than all emulation or mediation solutions running the same number of VMs.

In summary, vDPDK achieves a high performance compared to other non-pass-through modes, and stays performant even at a high number of VMs. The results of this test, therefore, align with our performance and scalability design goals. Furthermore, the measurable positive impact of offloading classification to hardware, configured in a device-independent manner, shows that we also achieve our generality design goal.

We can gain further information by comparing the measurements from the classification test in this section with the throughput measurements in Section 6.2.1. Packet rates in the classification tests are generally higher because only the RX path is used, while in the throughput tests, RX and TX are performed in parallel. For both pass-through modes and Qemu-vhost, if using hardware classification, packet rates during classification tests are almost exactly twice as high as in the throughput test. This suggests that RX and TX perform equally well.

On the other hand, when comparing packet rates of vMux E810 modes, we observe a packet rate increase by a factor of 10, from 115 kpps to 1.1 Mpps. This suggests a high overhead in the TX path. With vDPDK, on the other hand, performance barely differs. During the throughput tests, 12 Mpps are measured, and we measure 14 Mpps during the packet classification test. This suggests that the RX path of vDPDK is currently a bottleneck and a good candidate for future optimization efforts.

6.3 Non-DPDK benchmarks

While vDPDK performs well in the DPDK-based benchmarks above, we want to additionally evaluate its performance in typical network applications. However, since vDPDK is only usable with DPDK, and typical applications use the kernel network stack instead, we cannot directly use them. To make vDPDK accessible by regular applications, we essentially implement a simple user-space driver that creates a TAP device and uses DPDK to exchange packets between it and the vDPDK device.

This driver supports segmentation offloads and conserves resources by using RX interrupts instead of polling for new packets. Refer to Section 5.9 for further details. Note that this driver inherently has more overhead than in-kernel drivers due to frequent data copies and system calls necessary to operate the TAP interface. Therefore, the measurement results here should be seen more as a lower boundary for vDPDK, as we expect that DPDK-specific implementations of these tests will always perform equal or better.

6.3.1 TCP throughput

First, we measure TCP throughput with iperf3. On the VM, iperf3 is started in server mode. On the load generator, the iperf3 client is run in either regular, reverse, or bi-directional mode. In regular mode, the client, i. e. the load generator, sends a stream of data using TCP, therefore mainly stressing the RX path of the virtual device on the VM. In reverse mode, data is sent from the VM to the load generator, mainly stressing the TX path of the virtual device, and in bi-directional mode, data is sent from both server and client at the same time.

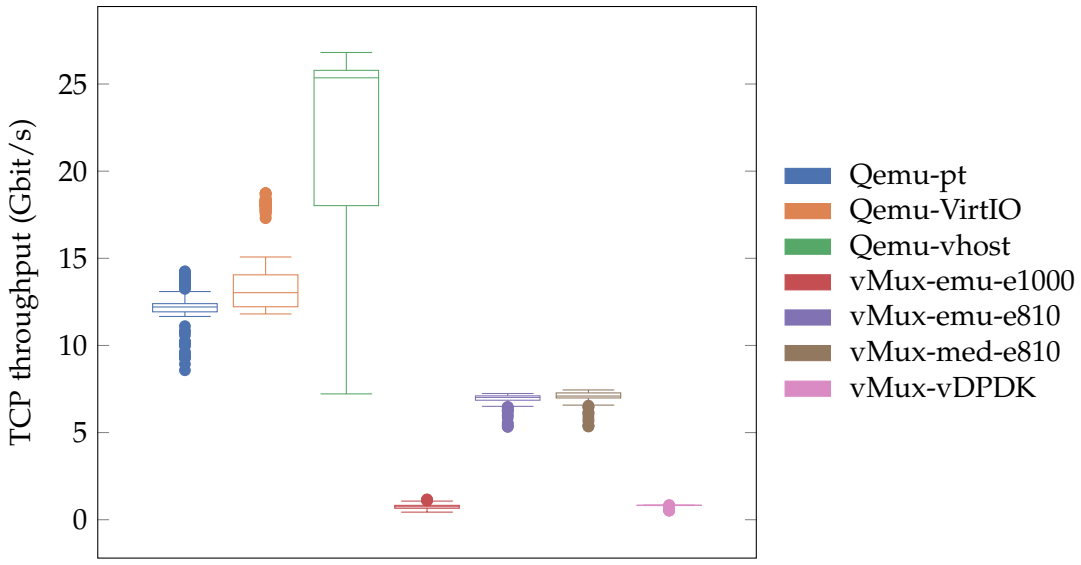


Figure 6.4: Results of iperf3 measurements in regular mode.

While the test is running, the current throughput is sampled at one-second intervals. In bi-directional mode, the throughput of the two streams is added together. Every test is run five times for 61 seconds. All samples are visualized as a box plot showing the overall throughput distribution.

Figure 6.4 shows the measurements obtained with iperf3 in its regular mode. Interestingly, the Qemu-vhost method achieves the overall highest performance with a median TCP throughput of 25 Gbit/s. Performance varies drastically, however, varying between 7 and 26 Gbit/s. Qemu-pt consistently reaches 12 Gbit/s with only a low number of outliers.

The vDPDK device only achieves a comparably low throughput of 800 Mbit/s. We assume this is caused by the following: First, as stated previously, the RX path of vDPDK seems to perform less well than TX. Second, no offloads are currently implemented for the RX path. This causes an especially large per-packet overhead in the vDPDK user-space driver, as every packet is copied to the TAP device individually, requiring a system call per packet.

The results of the reverse mode iperf3 measurements are shown in Figure 6.5. Here, pass-through performs best with a median throughput of 47 Gbit/s. Qemu-vhost again achieves a high but varying performance, with results between 17 and 38 Gbit/s, and a median throughput of 33 Gbit/s. The vDPDK device performs a bit worse, but still well, with a median throughput of 27 Gbit/s. This performance is mainly a result of consequently using TSO, and never segmenting the packet in software before it is

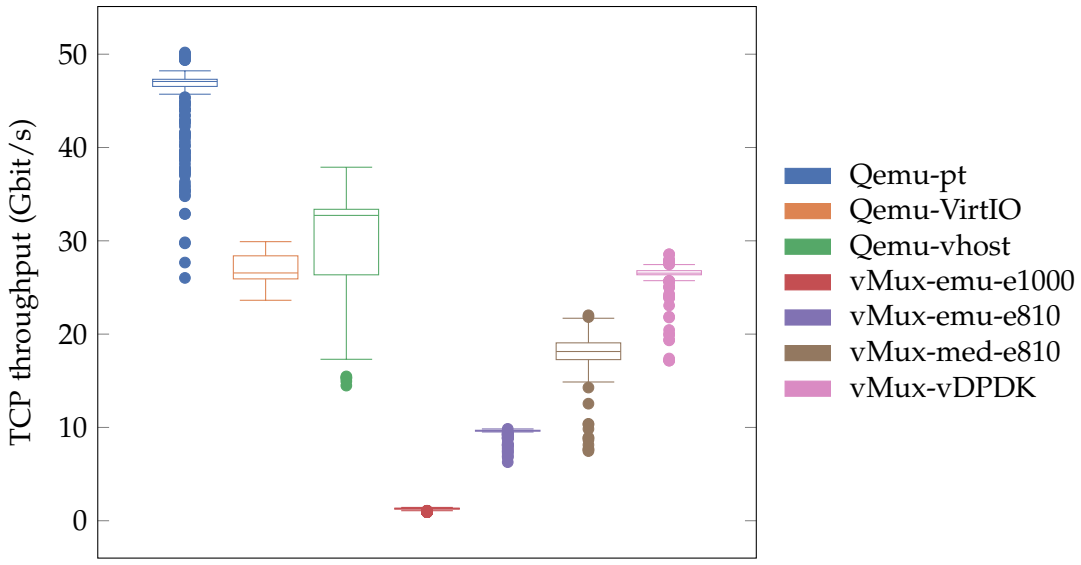


Figure 6.5: Results of iperf3 measurements in reverse mode.

segmented by the NIC.

The benefit of hardware TSO can also be seen by comparing the emulation and mediation modes of vMux's E810 emulator. In emulation mode, TSO is used by the guest, which reduces per-packet overhead during host–guest communication, but the packet is then segmented in software by the emulator. With this approach, a median throughput of 10 Gbit/s is reached. When using mediation mode, hardware TSO is used instead. This allows the same emulator to reach a median throughput of 18 Gbit/s.

Finally, measurements of the bi-directional mode of iperf3 are shown in Figure 6.6. The results here follow a similar pattern to the measurements taken in regular mode. The main notable difference is vMux-emu-e810, which saw a larger decrease in performance compared to the other virtual devices. With mediation enabled, this decrease is not visible, likely due to hardware TSO on the TX path. The vDPDK device performs at a median throughput of 600 Mbit/s. It is likely bottlenecked by its poor RX performance. Notably, on VMs with only one vCPU – as used for this experiment – the vDPDK user-space driver alternatively handles RX and TX on a single thread, which makes any bottlenecking even more apparent.

6.3.2 Cloud serving benchmark

To benchmark typical cloud workloads, we use a Redis benchmark from YCSB [6]. In this experiment, Redis shards are started on the load generator and loaded with data.

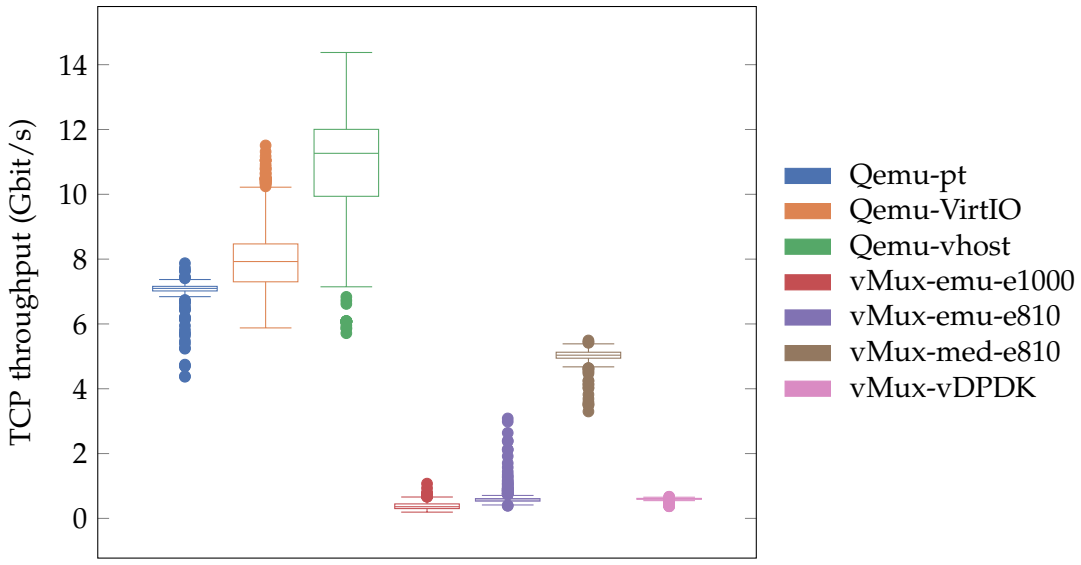


Figure 6.6: Results of iperf3 measurements in bi-directional mode.

Every VM then runs a pre-defined workload against these Redis shards. To evaluate performance, we measure the number of successful requests per second on every VM.

The results are displayed in Figure 6.7. While pass-through performs best with a single VM, vDPDK has the highest number of requests per second for every other number of VMs, scaling very well up to 32 VMs. To be specific, with one VM, pass-through reaches around 19 000 requests per second, while using vDPDK reduces performance by 21 % to approximately 15 000 requests per second. However, when scaling up to 32 VMs, per-VM performance is only reduced by 20 %, still reaching 12 000 requests per second.

At 64 VMs, a harsh drop to 6 000 requests per second can be observed. To explain this, we must consider the available resources of the VM host. The host has 48 available CPU cores, grouped into 8 clusters of 6 cores. With 32 VMs, every cluster runs 4 VMs – each with one vCPU – one RX thread, and one TX thread (as described in Section 4.3). This means every cluster has a task for each CPU core. With 64 VMs, every cluster must run 8 VMs and one thread for combined polling of RX and TX, exceeding the amount of available cores. The only reason this can still perform comparably well is because the vDPDK user-space driver running on the guests frequently parks its threads to wait for interrupts or new packets to send. This, in turn, helps with scheduling on the host.

For comparison, we perform the same experiment without using scalability features. Instead of scalable threading, one RX and one TX thread are used for each VM. Furthermore, instead of using the vDPDK user-space driver, we use FastClick to

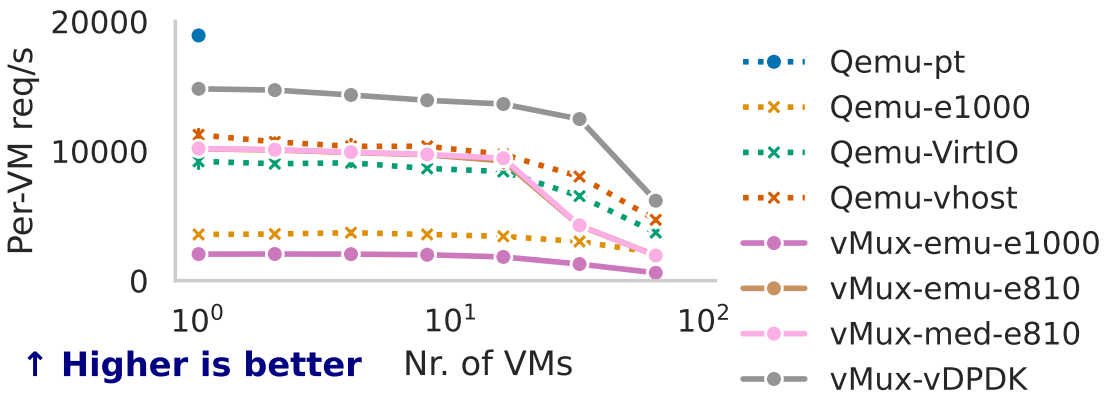


Figure 6.7: Results of the YCSB benchmark.

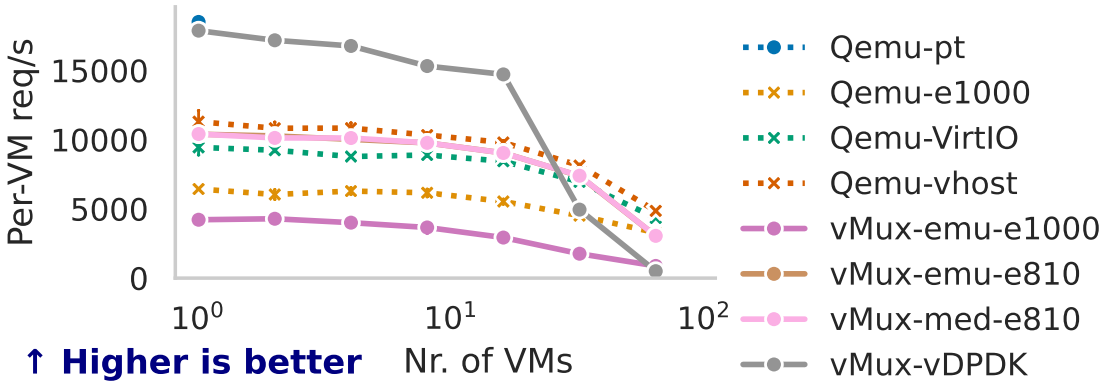


Figure 6.8: Results of the YCSB benchmark without interrupts or scalable threading.

exchange packets between the vDPDK device and a TAP interface. Unlike our driver, FastClick does not use interrupts and will busy-poll for new packets.

The results are shown in Figure 6.8. Due to aggressive polling, the number of requests per second achieved with vDPDK is actually higher at low VM counts. However, at 32 VMs, performance reduces drastically, while at 64 VMs, vDPDK performs worse than all other options, with around 500 requests per second. For comparison, with 32 VMs, every cluster now needs to run 4 VMs and 8 vMux polling threads on 6 cores. This shows the benefits of making scalability an explicit design goal.

In summary, vDPDK shows excellent performance at cloud serving loads, especially when considering that implementing such workloads in DPDK would likely yield further improvements. Furthermore, vDPDK displays excellent scaling behavior when increasing the number of VMs, performing well even when hardware resources are exhausted. Therefore, this test shows that we successfully achieve our performance and

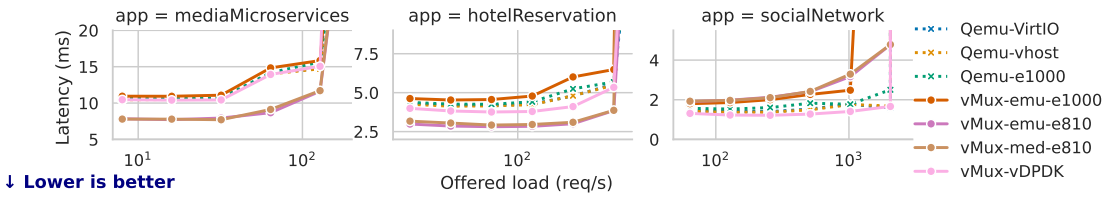


Figure 6.9: Latencies measured during the microservice benchmark.

scalability design goals.

6.3.3 Microservice benchmark

Finally, we evaluate the quality of cloud microservice architectures using the DeathStarBench [10] benchmark. We test all three available applications: a media service, a hotel reservation system, and a social network. In our experiment, we use a separate VM for each microservice and run a workload from the load generator. We measure the observed latency dependent on the current request load.

Note that with vMux, VMs cannot communicate with each other, as this would require vMux to essentially act as a software switch. Instead, any packets sent by a VM – even if addressed to another VM – are received by the load generator. To work around this, we use XDP to reflect any such packets back to vMux, where they will be received by the correct VM. This, of course, potentially increases observed latencies in this experiment when using vMux.

The results are displayed in Figure 6.9. In all applications, latencies stay fairly consistent until a threshold is reached, after which latencies increase drastically. With the exception of vMux-emu-e1000 in the social network application, all virtual devices share the same threshold for each application. This suggests that the threshold is likely determined by hardware resources instead of the network.

In the media service application, latencies increase drastically when exceeding only 128 requests per second. This suggests that this application requires a lot of computational resources. In this situation, the vMux E810 emulator shows the lowest latencies. Next is vDPDK, which performs very similarly to the QEMU emulators.

The hotel reservation application is less computationally intensive, performing well until the load exceeds 512 requests per second. Latencies are overall lower, with the vMux E810 emulator still performing best, and vDPDK coming second. And in the social network application, which reaches request rates up to 2048 before latencies increase, vDPDK achieves the overall lowest latencies.

These results suggest that vDPDK is especially affected by the computational load on the VM. This is likely due to the user-space driver that is required to use vDPDK

in this benchmark, as it needs to compete for processing resources with other tasks running on the system. Nevertheless, vDPDK achieves acceptable latencies similar to other virtual devices.

In summary, vMux scales well to the large number of VMs required for microservice architectures and offers high service quality comparable to other network virtualization approaches. This once again shows that vDPDK achieves its scalability design goal.

7 Related Work

In this thesis, we introduce vDPDK, built on vMux and DPDK, to improve networking in virtualized environments. This, however, is not the only possible approach. In this chapter, we review other work done in this area and compare their approaches to vDPDK.

7.1 Netmap-based solutions

There are various similar projects that use para-virtualization for IO, similar to vDPDK [13, 18, 15, 11]. For example, NetVM [15] also implements a new virtual device by integrating DPDK into QEMU. However, unlike vDPDK, it does not use DPDK on the guest, and instead implements a custom library. Most relevant is various work around the netmap framework.

Netmap is a framework for fast packet IO by Rizzo [34], and it fulfills a role similar to DPDK. The key difference is that netmap is supported as part of the in-kernel NIC driver, while DPDK uses a complete user-space driver. This means that DPDK only requires a driver giving it direct access to a NIC, while netmap functionality needs to be implemented for each kernel driver.

Garzarella, Lettieri, and Rizzo [11] introduce `ptnetmap`, a virtual network device that can be used to pass-through a netmap port from the host to a guest. This means that, similarly to vDPDK, this allows a netmap application on the guest to efficiently make use of netmap functions on the host. They show that, with this architecture, the VM can saturate a 10 Gbit link at 14.88 Mpps.

One goal of vMux is to be able to scale to a large number of VMs, utilizing hardware offloads to direct incoming packets to the correct VM. With `ptnetmap`, this approach is not possible, as the host netmap port cannot be shared between multiple VMs. The intended alternative is using VALE [35], a netmap-based software switch. This switch connects to the physical NIC via netmap, and can be accessed efficiently by VMs via `ptnetmap`. They show that they can reach packet rates of up to 24 Mpps in direct guest-to-host communication, but to our knowledge, there are no measurements for communication between a guest and an external machine connected via VALE, or measurements demonstrating the scaling behavior of this approach.

One advantage of the vMux approach is that hardware capabilities can be dynamically discovered and used by multiple virtual machines. This is not possible if the physical NIC is only accessed indirectly via a software switch.

7.2 Software switches

When using VMs with emulated network devices, the VMs are usually connected to the wider network via a software switch [11, 9]. One notable software switch is Open vSwitch [32], which uses a kernel module to receive and transmit packets, and a user-space process to make complex forwarding decisions. Other modern software switches are inspired by the modular architecture of Click [17]. These include switches like VPP [2], FastClick [3], and BESS [12, 29]. They directly access physical NICs via DPDK or the previously described netmap interface.

vMux focuses on giving VMs access to offloads supported by a single physical NIC and does not implement a software switch. To "switch" packets to the correct VM, vMux uses hardware-offloaded queue steering instead, similar to SR-IOV setups. The main disadvantage of this approach is that direct inter-VM communication is not possible. While vMux can be run on top of a software switch, this would be at the cost of losing advanced offload support. To enable inter-VM communication without impacting performance and offload support, a hardware switch with support for hairpin switching can be used instead [31].

7.3 Hypervisor-based virtual networking

In vDPDK, any packet processing is done by a DPDK application on the guest, while actual IO is performed by a separate thread on the host. While we focus on ensuring the scalability of this approach, performance still suffers once the number of VMs grows past the number of available CPU cores.

Yasukata, Huici, Maffione, et al. [38] call this a "split" model, because VM processing and corresponding IO are split between two threads. They argue that this approach is inefficient as it is "essentially a static allocation of resources". As an alternative, they suggest a "merge" model in which VM processing and IO are executed on the same CPU core, and the scheduler can dynamically change the allocation of processing resources.

Typically, such a model would require frequent context switches, reducing performance. As a solution, Yasukata, Huici, Maffione, et al. [38] introduce HyperNF, which moves IO code into the hypervisor and calls it from the context of a VM via hypercall.

For low packet sizes, this approach achieves packet rates close to 15 Mpps, 60 % higher than the split model.

This approach requires support by the hypervisor and is therefore not directly suitable for vMux and vDPDK, which are pure user-space implementations. However, adopting a similar merge model could be beneficial for scalability.

7.4 Software-hardware co-designed virtualization

The main issue with using SR-IOV for VM networking is its rigid resource management and limits on the maximum amount of supported virtual devices. Implementing SR-IOV requires complex hardware designs to realize virtualization logic in hardware.

Z. Zhang, J. Chen, Ying, et al. [39] present HD-IOV, a functional alternative to SR-IOV, that uses software-hardware co-design to allow more dynamic resource management and scale to a larger number of VMs. The key idea is to implement all management logic in software while utilizing the IOMMU and additional hardware modifications to provide isolation between virtual devices. They measure performance comparable to SR-IOV and a maximally 2.96x higher device count.

The main drawback of HD-IOV, compared to vMux, is that it requires hardware modifications. Its main advantage, however, is very high and stable performance.

8 Future Work

While the current implementation of vDPDK works well, it is not feature-complete. Various additions and enhancements are available for possible future work.

8.1 Targeted optimization

In its current state, vDPDK is performant by design, not because of implementation efforts. While care was taken to ensure the implementation performs well, no targeted optimizations were performed.

In future work, targeted profiling of various parts of vDPDK could be performed to identify current bottlenecks. With these results, optimization techniques could be applied where relevant to increase the overall performance of vDPDK. For example, in Chapter 6, we specifically identify the RX path as performing comparably slow, providing a good target for first optimizations.

8.2 Increased DPDK feature support

Currently, the vDPDK driver only implements a fraction of all the operations supported by DPDK. While this is a valid approach, as no driver is expected to support all operations, ideally, most operations should be forwarded to vMux to enable the use of all operations supported by the host NIC.

In the future, more of these operations could be implemented, following the current control plane design. This requires determining how to serialize parameters and how to ensure other VMs cannot be directly affected by this operation.

A promising target is increasing support of the `rte_flow` API. Currently, support is very limited, and it is only enough to demonstrate feasibility and run our experiments. Increasing support would allow users to offload various RX packet processing tasks, an area in which the current implementation is lacking overall.

8.3 Zero-copy data plane

While basic support for zero-copy TX exists, it comes with various unsolved problems described in Section 4.1.3. Additionally, no such support is implemented for the RX path.

Completing the zero-copy implementation in future work would require solving the existing DMA mapping invalidation problem, which likely requires modifying either libvfio-user or DPDK driver code. Additionally, descriptor management overhead would have to be reduced, for example, by simplifying the descriptor layout or adding a layer of indirection to the ring.

9 Conclusion

In this thesis, we design and implement vDPDK, a para-virtualized device for vMux. It can be used by DPDK applications in VMs to achieve high networking performance at low latencies, while also being able to access advanced offloading features of the backing physical NIC. This is realized by designing vDPDK as a thin layer between DPDK running on the guest and the DPDK backend running in vMux.

In Chapter 3 we designate three design goals: performance, generality, and scalability. We use various experiments to evaluate whether vDPDK fulfills these goals. DPDK throughput tests show that high performance can be achieved with vDPDK, while experiments using our interrupt-based user-space driver demonstrate scalability. Because none of the experiments use code that is specialized for vDPDK or the physical NIC, we also achieve the goal of generality.

All source code produced for this thesis is available at `https://github.com/vmuxIO`. This includes source code for vMux, our fork of DPDK, measurement scripts, and reproducible development environments. Specifically, this thesis describes version 0.0.13 of vMux and its recorded dependencies.

Abbreviations

VM virtual machine

VMM virtual machine monitor

NIC network interface controller

OS operating system

SR-IOV single-root I/O virtualization

DPDK the Data Plane Development Kit

DMA direct memory access

TSO TCP segmentation offload

TX transmission

RX reception

List of Figures

List of Tables

Bibliography

[1] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. "Firecracker: Lightweight Virtualization for Serverless Applications." In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[2] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi. "High-Speed Software Data Plane via Vectorized Packet Processing." In: *IEEE Communications Magazine* 56.12 (2018), pp. 97–103. DOI: 10.1109/MCOM.2018.1800069.

[3] C. S. Barbette Tom and M. Laurent. "Fast userspace packet processing." In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2015, pp. 5–16. DOI: 10.1109/ANCS.2015.7110116.

[4] F. Bellard and the QEMU team. *QEMU*. URL: https://www.qemu.org.

[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: *2nd Symposium on Networked Systems Design and Implementation (NSDI) 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. Ed. by A. Vahdat and D. Wetherall. USENIX, 2005. URL: http://www.usenix.org/events/nsdi05/tech/clark.html.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking cloud serving systems with YCSB." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152.

[7] DPDK developers. *DPDK Programmer's Guide. Generic flow API*. 2024. URL: https://doc.dpdk.org/guides/prog_guide/ethdev/flow_offload.html.

[8] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. "MoonGen: A Scriptable High-Speed Packet Generator." In: *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan, Oct. 2015.

[9] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle. "Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis." In: *Journal of Network and Systems Management* 26.2 (Apr. 2018), pp. 314–338. ISSN: 1573-7705. DOI: 10.1007/s10922-017-9417-0. URL: https://doi.org/10.1007/s10922-017-9417-0.

[10] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18. ISBN: 9781450362405. DOI: 10.1145/3297858.3304013. URL: https://doi.org/10.1145/3297858.3304013.

[11] S. Garzarella, G. Lettieri, and L. Rizzo. "Virtual device passthrough for high speed VM networking." In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2015, pp. 99–110.

[12] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. *SoftNIC: A Software NIC to Augment Hardware*. Tech. rep. UCB/EECS-2015-155. May 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[13] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky. "Efficient and Scalable Paravirtual I/O System." In: *Proceedings of the 2013 USENIX Annual Technical Conference, USENIX ATC 2013, San Jose, CA, USA, June 26-28, 2013*. Ed. by A. Birrell and E. G. Sirer. USENIX Association, 2013, pp. 231–242. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/har%5C%E2%5C%80%5C%99el.

[14] M. R. Hines, U. Deshpande, and K. Gopalan. "Post-copy live migration of virtual machines." In: *SIGOPS Oper. Syst. Rev.* 43.3 (July 2009), pp. 14–26. ISSN: 0163-5980. DOI: 10.1145/1618525.1618528. URL: https://doi.org/10.1145/1618525.1618528.

[15] J. Hwang, K. K. Ramakrishnan, and T. Wood. "NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms." In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*. Ed. by R. Mahajan and I. Stoica. USENIX Association, 2014, pp. 445–458. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang.

[16] Intel. *Intel Ethernet Controller E810. Datasheet*. Version Revision 2.8. 2024. URL: https://cdrdv2.intel.com/v1/dl/getContent/613875 (visited on 05/17/2025).

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. "The click modular router." In: *ACM Trans. Comput. Syst.* 18.3 (Aug. 2000), pp. 263–297. ISSN: 0734-2071. DOI: 10.1145/354871.354874. URL: https://doi.org/10.1145/354871.354874.

[18] Y. Kuperman, E. Moscovici, J. Nider, R. Ladelsky, A. Gordon, and D. Tsafrir. "Paravirtual Remote I/O." In: *SIGPLAN Not.* 51.4 (Mar. 2016), pp. 49–65. ISSN: 0362-1340. DOI: 10.1145/2954679.2872378. URL: https://doi.org/10.1145/2954679.2872378.

[19] Linux Foundation. *Data Plane Development Kit*. URL: https://www.dpdk.org.

[20] J. Liu. "Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support." In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470365.

[21] T. Makatos, S. Ingle, F. Franciosi, and libvfio-user contributors. *libvfio-user*. URL: https://github.com/nutanix/libvfio-user.

[22] T. Makatos, S. Ingle, F. Franciosi, and libvfio-user contributors. *vfio-user Protocol Specification*. Version 0.9.2. URL: https://github.com/nutanix/libvfio-user/blob/master/docs/vfio-user.rst (visited on 05/19/2025).

[23] E. P. Martín. *Deep dive into Virtio-networking and vhost-net*. July 12, 2019. URL: https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net.

[24] E. P. Martín. *Virtqueues and virtio ring: How the data travels*. July 8, 2020. URL: https://www.redhat.com/en/blog/virtqueues-and-virtio-ring-how-data-travels.

[25] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. "Snap: a microkernel approach to host networking." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 399–413. ISBN: 9781450368735. DOI: 10.1145/3341301.3359657. URL: https://doi.org/10.1145/3341301.3359657.

[26] M. Nelson, B. Lim, and G. Hutchins. "Fast Transparent Migration for Virtual Machines." In: *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005, pp. 391–394. URL: http://www.usenix.org/events/usenix05/tech/general/nelson.html.

[27] Nvidia. *Nvidia ConnectX-7 Datasheet*. 2021. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband-adapters/infiniband-connectx7-data-sheet.pdf (visited on 05/19/2025).

[28] P. Okelmann, M. Misono, et al. "vMux: A Unified Network Device Virtualization Architecture." Paper is not yet published. 2025.

[29] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. "E2: a framework for NFV applications." In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. Ed. by E. L. Miller and S. Hand. ACM, 2015, pp. 121–136. DOI: 10.1145/2815400.2815423. URL: https://doi.org/10.1145/2815400.2815423.

[30] PCI-SIG. *Single Root I/O Virtualization and Sharing Specification*. 2010. URL: https://pcisig.com/specifications.

[31] J. Pettit, J. Gross, B. Pfaff, M. Casado, and S. Crosby. "Virtual switching in an era of advanced edges." In: *2nd Workshop on Data Center - Converged and Virtual Ethernet Switching (DC-CAVES)*. ITC 22. Sept. 6, 2010.

[32] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. "The Design and Implementation of Open vSwitch." In: *login Usenix Mag.* 40.2 (2015). URL: https://www.usenix.org/publications/login/apr15/pfaff.

[33] H. Raj and K. Schwan. "High performance and scalable I/O virtualization via self-virtualized devices." In: *Proceedings of the 16th International Symposium on High Performance Distributed Computing*. HPDC '07. Monterey, California, USA: Association for Computing Machinery, 2007, pp. 179–188. ISBN: 9781595936738. DOI: 10.1145/1272366.1272390. URL: https://doi.org/10.1145/1272366.1272390.

[34] L. Rizzo. "netmap: a novel framework for fast packet I/O." In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 101–112.

[35] 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*. 2012, pp. 61–72.

[36] The kernel development community. *The Definitive KVM (Kernel-based Virtual Machine) API Documentation*. 2025. URL: https://www.kernel.org/doc/html/v6.14/virt/kvm/api.html.

[37] M. S. Tsirkin and C. Huck, eds. *Virtual I/O Device (VIRTIO) Version 1.1. OASIS Committee Specification 01*. Apr. 11, 2019. URL: https://docs.oasis-open.org/virtio/virtio/v1.1/cs01/virtio-v1.1-cs01.html.

[38] K. Yasukata, F. Huici, V. Maffione, G. Lettieri, and M. Honda. "HyperNF: building a high performance, high utilization and fair NFV platform." In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, California: Association for Computing Machinery, 2017, pp. 157–169. ISBN: 9781450350280. DOI: 10.1145/3127479.3127489. URL: https://doi.org/10.1145/3127479.3127489.

[39] Z. Zhang, J. Chen, B. Ying, Y. Cao, L. Liu, J. Li, X. Zeng, J. Wang, W. Li, and H. Guan. "HD-IOV: SW-HW Co-designed I/O Virtualization with Scalability and Flexibility for Hyper-Density Cloud." In: *Proceedings of the Nineteenth European Conference on Computer Systems*. EuroSys '24. Athens, Greece: Association for Computing Machinery, 2024, pp. 834–850. ISBN: 9798400704376. DOI: 10.1145/3627703.3629557. URL: https://doi.org/10.1145/3627703.3629557.

[40] Z. Zhang, C. Xia, C. Liang, J. Li, C. Yu, T. Bie, R. Martin, D. Dan, X. Wang, Y. Liu, and H. Guan. "Un-IOV: Achieving Bare-Metal Level I/O Virtualization Performance for Cloud Usage With Migratability, Scalability and Transparency." In: *IEEE Transactions on Computers* 73.7 (2024), pp. 1655–1668. DOI: 10.1109/TC.2024.3375589.