Technology Guide

Intel® Data Streaming Accelerator (DSA) - Accelerating DPDK Vhost

Authors

Jiayu Hu

Cheng Jiang

Xuan Ding

Xingguang He

1 Introduction

VirtIO[1] is a virtualized interface standard for virtual machines (VMs) to access devices such as network devices and block devices. A VirtIO instance consists of a backend running on the host machine, and a frontend, which is present in the guest VM. Data Plane Development Kit (DPDK) Vhost library[2] is a VirtIO backend implementation for VirtIO network devices, and it is widely used in software switches for VMs to receive and transmit packets at high speeds. Although the DPDK Vhost library is well optimized with various techniques (for example, loop unrolling and using inline functions to avoid function call overheads), a high proportion of core cycles is spent on packet copying when the packet size is large, thus causing Vhost packet transmit and receive rates to decrease rapidly.

Intel® Data Streaming Accelerator is a high-performance data copy and transformation accelerator in the 4th Gen Intel® Xeon® Scalable processor (formerly code named Sapphire Rapids[3]). Providing high bandwidth and low latency for data movement, it is an ideal accelerator to accelerate the performance of DPDK Vhost library.

We offload packet copies to Intel Data Streaming Accelerator in the DPDK Vhost library, where the Vhost core can eliminate copying packets. We also propose an asynchronous offloading pipeline to enable applications to hide Intel Data Streaming Accelerator copy latency with higher level functions processed by the core. Our experiments show that Intel Data Streaming Accelerator can bring considerable packet forwarding rate improvement for the DPDK Vhost library when the packet size is beyond 256 bytes.

This document is part of the Network Transformation Experience Kits.

[1] https://wiki.osdev.org/Virtio

[2] https://doc.dpdk.org/guides/prog_guide/vhost_lib.html

[3] https://en.wikipedia.org/wiki/Sapphire_Rapids

Table of Contents

Figures

Tables

Document Revision History

Revision	Date	Description
001	December 2022	Initial release.

1.1 Terminology

Table 1. Terminology

Abbreviation	Description
API	Application Programming Interface
DPDK	Data Plane Development Kit
Intel DSA	Intel Data Streaming Accelerator
PMD	Polling Mode Driver
VM	Virtual Machine
WQ	Work Queue
DWQ	Dedicated Work Queue

1.2 Reference Documentation

Table 2. Reference Documents

Reference	Source
VirtIO Spec Version 0.95	https://ozlabs.org/~rusty/virtio-spec/virtio-0.9.5.pdf
VirtIO Spec Version 1.1	https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html
Intel Data Streaming Accelerator	https://01.org/blogs/2019/introducing-intel-data-streaming-accelerator

2 Overview

The DPDK Vhost library is a fast VirtIO backend implementation for VirtIO networking devices in user-space. It is widely used virtual interface in Open vSwitch for packet I/O to VMs. In the following sections of this document, we will use the term "DPDK Vhost" to refer to the DPDK Vhost library.

2.1 Data Plane Development Kit (DPDK) Vhost

DPDK Vhost consists of a set of control path and data path Application Programming Interfaces (APIs). Control path APIs are used for mapping VM memory, setting up the host to guest virtqueues, and negotiating features with the VirtIO frontend. Data path APIs are used to send/receive packets to/from VirtIO frontend, implemented by the two key data plane operations, enqueue and dequeue, respectively. On enqueue, Vhost transmits packets to the VirtIO frontend by inserting packets into a virtqueue, and on dequeue, Vhost receives packets from the VirtIO frontend by taking out packets from a virtqueue. Thus, at least two VirtIO virtqueues are required for bi-directional data transport between Vhost and VirtIO frontend.

One enqueue or dequeue operation can be divided into the following three sub-operations, as shown in Figure 1.

1. Firstly, available descriptors that describe buffers in VM memory are fetched from the virtqueue. Then, from those descriptors, the buffer addresses for later packet operations are read. For enqueue, these available descriptors point to empty buffers used to store packets in future; for dequeue, the available descriptors point to packet buffers sent by VirtIO frontend.
2. Secondly, packets are copied from the host to the obtained buffers for enqueue, or from the obtained packet buffers to the host for dequeue.
3. Thirdly, "used" descriptors are written back to the virtqueue and a notification is sent to the VM.

These three sub-operations are performed by the DPDK Vhost core in order. After they are completed, the DPDK Vhost enqueue or dequeue function returns to the user application and the core can perform higher-level functions for the application.

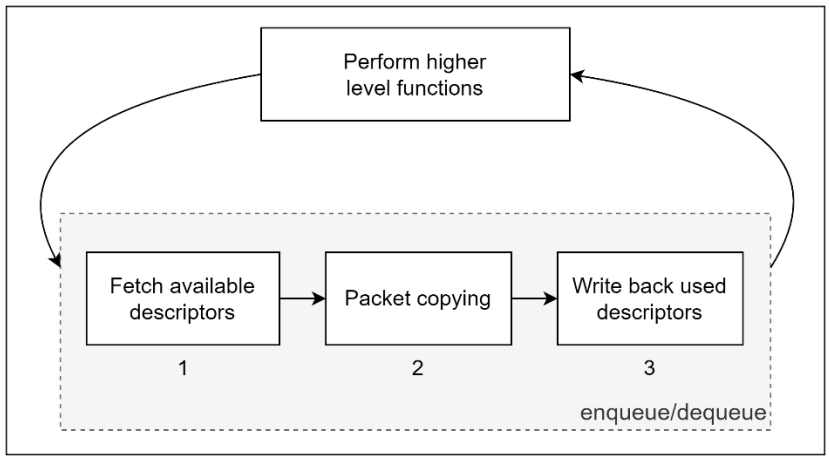

Figure 1. Enqueue/Dequeue Operation Workflow

DPDK Vhost supports two virtqueue formats: split virtqueue and packed virtqueue, which are introduced in VirtIO Spec versions 0.95[4] and 1.1[5] respectively. The split virtqueue consists of three parts, a descriptor table, an available ring, and a used ring. Unlike the split virtqueue, the packed virtqueue merges the three parts of the virtqueue into one and it only contains one descriptor ring with different descriptor formats. In the following paragraphs, we will use the split virtqueue to demonstrate the workflow of enqueue/dequeue operation.

A split virtqueue consists of a descriptor table, an available ring, and a used ring. The descriptor table is an array of descriptors describing packet buffers, and each available/used ring is an index array pointing to descriptors in the descriptor table. The difference between the available ring and the used ring is that the available ring stores the indices of descriptors available to Vhost, but the used ring stores indices of descriptors available to the VirtIO frontend. Each ring also has a tail pointer that is used to indicate where the VirtIO frontend, for avail ring, or Vhost, for used ring, would put the next descriptor index, and the tail pointer is increased after filling descriptor indices to the ring. The tail pointer is called *avail_idx* and *used_idx* for the available ring and used ring, respectively.

Figure 2 shows an example of sending four packet buffers by the VirtIO frontend and Vhost dequeue them from the split virtqueue. The VirtIO frontend stores packet buffer information in the first four descriptors, that is, from index 0 to index 3 in the descriptor table. It then fills 0~3 in the available ring and increases *avail_idx* by four. On the host side, Vhost reads the available ring to get the four packet buffer addresses, then copies packets from VM memory into host memory. After finishing copying the four packet buffers, Vhost writes back 0~3 to the used ring and increases the *used_idx* by four to notify the frontend that the four packet buffers have been received by Vhost.

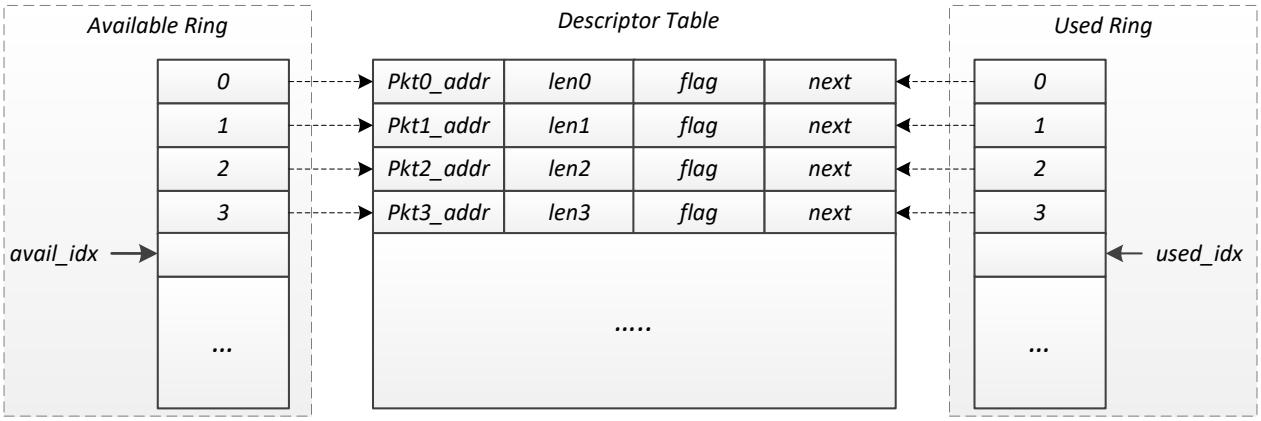

Figure 2. Example of Split Virtqueue Operation

[4] https://ozlabs.org/~rusty/virtio-spec/virtio-0.9.5.pdf
[5] https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html

2.2 Intel® Data Streaming Accelerator

Intel Data Streaming Accelerator[6] (Intel DSA) is a high-performance data copy and transformation accelerator that is in 4th Gen Intel Xeon Scalable processors. Work queues (WQs) are on-device storage to contain descriptors that have been submitted to device and are used by applications to assign work to the Intel DSA. Batch descriptor allows software to submit multiple work descriptors using a single work submission operation and can potentially improve overall throughput, especially for small transfer sizes. In the DPDK Intel DSA driver[7], Dedicated Work Queue (DWQ) and Batch descriptor are used.

2.3 Motivation

As an efficient VirtIO backend implementation, DPDK Vhost is widely used in software switches, like Open vSwitch. Although DPDK Vhost is well optimized with various techniques (for example, loop unroll and inline function), with increasing packet sizes, packet copying in Vhost starts to limit performance. We use the typical DPDK application TestPMD as a software switch in the host and it connects to one VM via a Vhost PMD[8] port. We study the impact of packet copies in the Vhost PMD port to the host TestPMD. The experiment configuration is the same as that of Section 4.1. By varying the packet size, we measure the packet forwarding rate for the host TestPMD and the fractions of core cycles spent in switching, virtqueue operation, and packet copying in Vhost via Linux Perf.

The results are presented in Figure 3. We can see that the larger packet size causes a noticeable increase of core cycles spent in Vhost packet copy. At 1518 bytes, 59% core cycles are spent in Vhost packet copy. We can also see that larger packet size implies lower packet forwarding rate of TestPMD. This is because the amount of core cycles spent in Vhost packet copying increases with packet sizes and Vhost packet copy becomes the bottleneck in TestPMD.

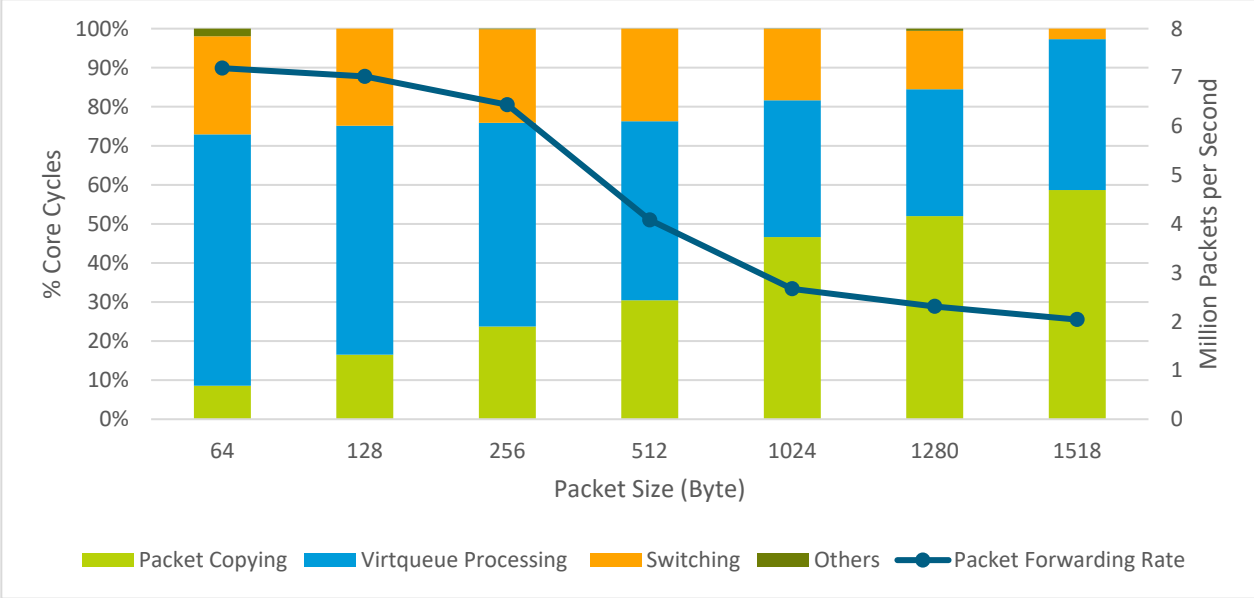

Figure 3. Core Cycle Distribution in Host TestPMD

3 DPDK Vhost with Intel® DSA Acceleration

3.1 Overview

Figure 4 shows the design overview of DPDK Vhost with Intel DSA acceleration. As discussed in section 2.1, one enqueue/dequeue operation can be divided into two sub-operations: virtqueue processing and packet copying. Virtqueue processing is done fast by the core. This is because the descriptor size is small, and the core is efficient at performing reads/writes on small data. Compared with virtqueue processing, packet copying is more expensive, especially when the packet size is large. As a result, we offload all packet copies to the Intel DSA (shown in blue arrow) but leave virtqueue operations to the core (shown in green arrow) for both enqueue and dequeue operations.

[6] https://01.org/blogs/2019/introducing-intel-data-streaming-accelerator

[7] https://doc.dpdk.org/guides/dmadevs/idxd.html

[8] https://doc.dpdk.org/guides/nics/vhost.html

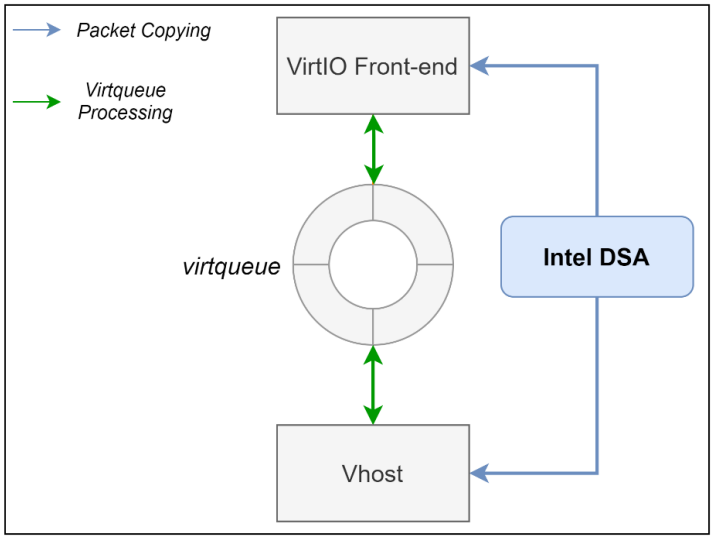

Figure 4. Design Overview of DPDK Vhost with Intel® DSA Acceleration

Besides the gain from high Intel DSA data movement bandwidth, we also propose an asynchronous offloading pipeline to enable applications to hide Intel DSA copy latency with higher level functions. The asynchronous offloading pipeline is shown in Figure 5.

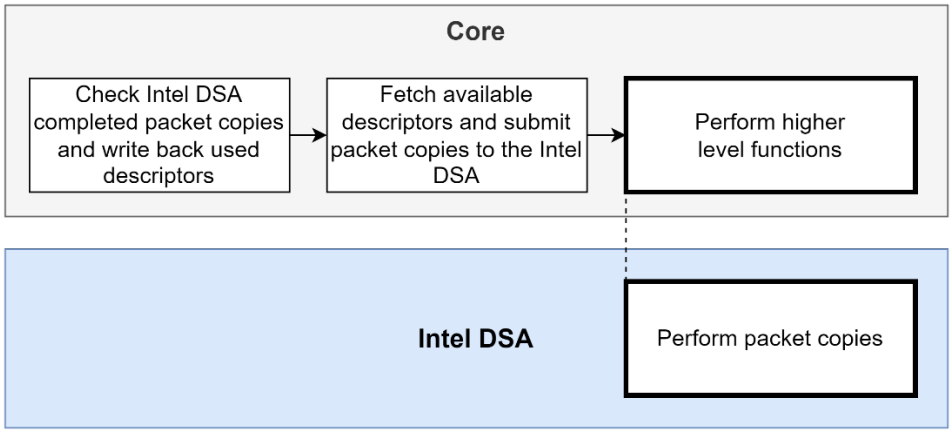

Figure 5. Asynchronous Offloading Pipeline

First, the core checks that Intel DSA completed packet copies submitted previously and writes back the corresponding used descriptors to the virtqueue. In this stage, the core does not wait for the Intel DSA to complete all submitted packet copies. Second, the core submits new packet copies to the Intel DSA after obtaining available descriptors from the virtqueue. Then, the Intel DSA starts performing copy jobs immediately. In the meanwhile, the core can perform higher level functions for the application. Packet copying by the Intel DSA and higher level function processed by the core are performed in parallel. Therefore, the application can hide Intel DSA copy latency with its higher level functions and achieve higher throughput.

3.2 Virtqueue and Intel® DSA

3.2.1 Virtqueue and Intel® DSA Work Queue Mapping

DPDK Vhost with Intel DSA acceleration supports M: N mapping between virtqueues and Intel DSA WQs. Specifically, one Intel DSA WQ can be used by multiple virtqueues and one virtqueue can offload copies to multiple Intel DSA WQs at the same time. Figure 6 shows an example for the M: N mapping between virtqueue and Intel DSA WQ. WQ2 is shared by virtqueue2, virtqueue3, and virtqueue4. virtqueue2 uses two Intel DSA WQs at the same time.

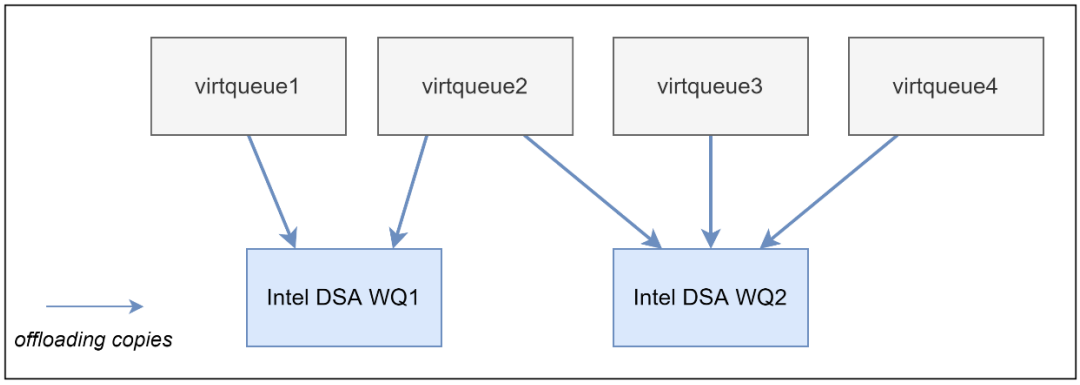

Figure 6. Example of Virtqueue and Intel® DSA WQ Mapping

3.2.2 Ordering

Vhost processes descriptors in the order they appear in the virtqueue. However, the Intel DSA might complete copies out of order, especially when offload copies of a virtqueue to multiple Intel DSA WQs. It could cause packets out-of-order after dequeue/enqueue and result in severe performance issues for applications.

To guarantee the packet ordering, every virtqueue is assigned with a reordering array to track if packets are completed by the Intel DSA. After all copies of a packet are completed by the Intel DSA, the corresponding element in the reordering array is marked with DONE flag. The Vhost core traverses the array and writes descriptors to virtqueue for the packets with DONE flag; once meets an uncompleted packet, the Vhost core stops writing back descriptors for the one and the subsequent packets. Therefore, although copies could be completed in any order by the Intel DSA, the reordering array can guarantee the ordering of packets.

Figure 7 shows how multiple Intel DSA WQs work in the example of section 2.1. After getting the four available descriptors in Vhost, the core submits four source and destination address pairs to four Intel DSA WQs (WQ0, WQ1, WQ2, and WQ3) respectively. Currently, pkt0, pkt2, and pkt3 are done, but Intel DSA WQ2 does not finish pkt1. In the reordering array, the second element is not marked with DONE flag, so the core only returns pkt0 to the user application and writes back index 0 to the used ring. After pkt1 is completed by Intel DSA WQ2, the core will return pkt1, pkt2, and pkt3 to the user application and write back the index 1~3 to the used ring.

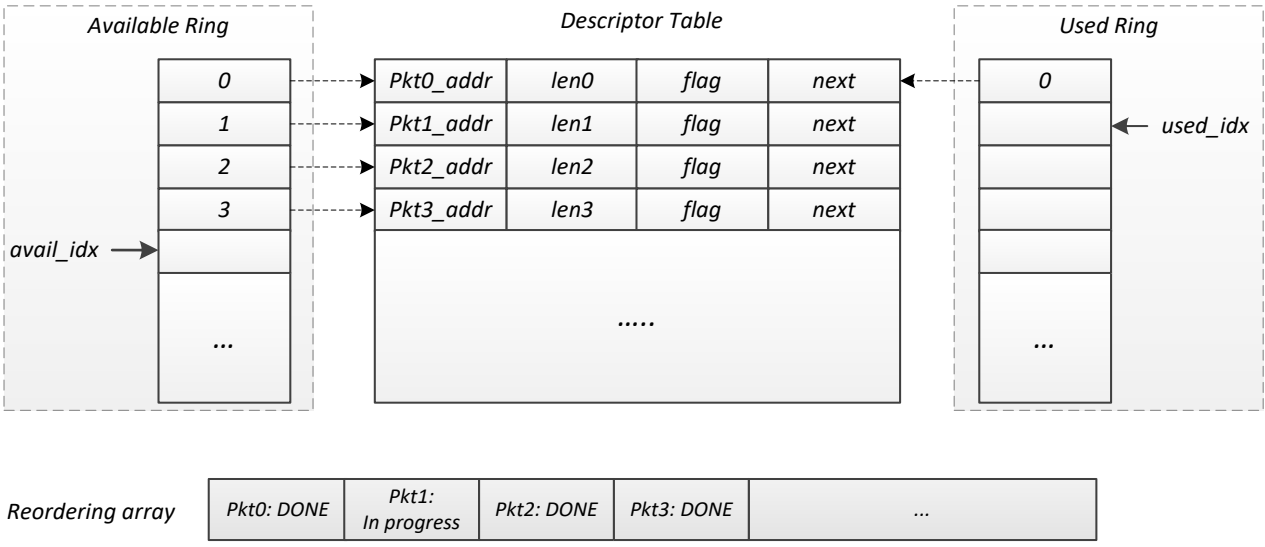

Figure 7. Virtqueue with Multiple Intel® DSA WQs Example

3.3 DPDK Vhost Asynchronous APIs

Compared with software copy, offloading packet copies to the Intel DSA introduces higher latency for enqueue/dequeue operations. DPDK Vhost provides a set of asynchronous APIs for applications to hide Intel DSA copy latency with the high-level application logic done by the core.

DPDK Vhost leverages the DPDK dmadev library that is a generic DMA engine library to operate the hardware DMA device. As a result, besides the Intel DSA, it can support other DMA engines, like Crystal Beach DMA. The dmadev library exposes an Intel DSA WQ as a DMA device instance with one virtual channel. In the following section, we will use the term "DMA device instance" and "virtual channel" instead of Intel DSA WQ to demonstrate the use of DPDK Vhost asynchronous APIs.

The DPDK Vhost asynchronous API includes control path and data path APIs. Before starting data path, the control path API is used to enable DMA accelerated data path for virtqueues via three steps.

1. Firstly, tell Vhost all DMA device instances and virtual channels that will be used in the data path via rte_vhost_async_dma_configure().
2. Secondly, notify Vhost to set up environment for the Vhost port that will use DMA accelerated data path via passing RTE_VHOST_USER_ASYNC_COPY in rte_vhost_driver_register().
3. Thirdly, register the virtqueue that will use DMA accelerated data path to Vhost via calling rte_vhost_async_channel_register() inside the Vhost callback function vring_state_changed(), when the virtqueue is enabled. Then user applications can call asynchronous data path APIs on the virtqueue that successfully registers.

Asynchronous data path APIs include enqueue and dequeue operations. Transmitting packets to VM via the asynchronous enqueue API requires two steps.

1. Firstly, fetch available descriptors from the virtqueue and submit packet copies to a given virtual channel of DMA device instance via rte_vhost_submit_enqueue_burst(). Then the DMA device starts to copy packets.
2. Secondly, check completed packet copies on the given virtual channel of DMA device instance and write back the corresponding used descriptors to virtqueue via rte_vhost_poll_enqueue_completed(). Then the VirtIO frontend is notified of successfully transmitted packets by the DMA device.

The asynchronous enqueue API requires applications to call rte_vhost_poll_enqueue_completed() appropriately; otherwise, the VirtIO frontend will not receive packets, even if the packets have been copied into the VM memory by the DMA device. In dequeue, there is only one API, rte_vhost_try_dequeue_burst(). It submits new packet copies to a given virtual channel of DMA device instance; in addition, it checks completed packet copies on the DMA virtual channel that are submitted previously and returns transmission completed packets to applications.

When the virtqueue is disabled or the Vhost device is destroyed, the control path API can be used to disable DMA accelerated data path for the virtqueue via three steps.

1. Firstly, clear in-flight packets in the virtqueue. This can be done by calling rte_vhost_clear_queue() inside the Vhost callback function destroy_device() or calling rte_vhost_clear_queue_thread_unsafe() inside the Vhost callback function vring_state_changed().
2. Secondly, unregister the virtqueue from DMA accelerated data path. This can be done by calling rte_vhost_async_channel_unregister() inside the Vhost callback function destroy_device() or calling rte_vhost_async_channel_register_thread_unsafe() inside the Vhost callback function vring_state_changed().
3. Thirdly, unconfigure DMA device instances and virtual channels that will not be used in the data path via rte_vhost_async_dma_unconfigure().

Figure 8 gives an example of using DPDK Vhost asynchronous APIs.

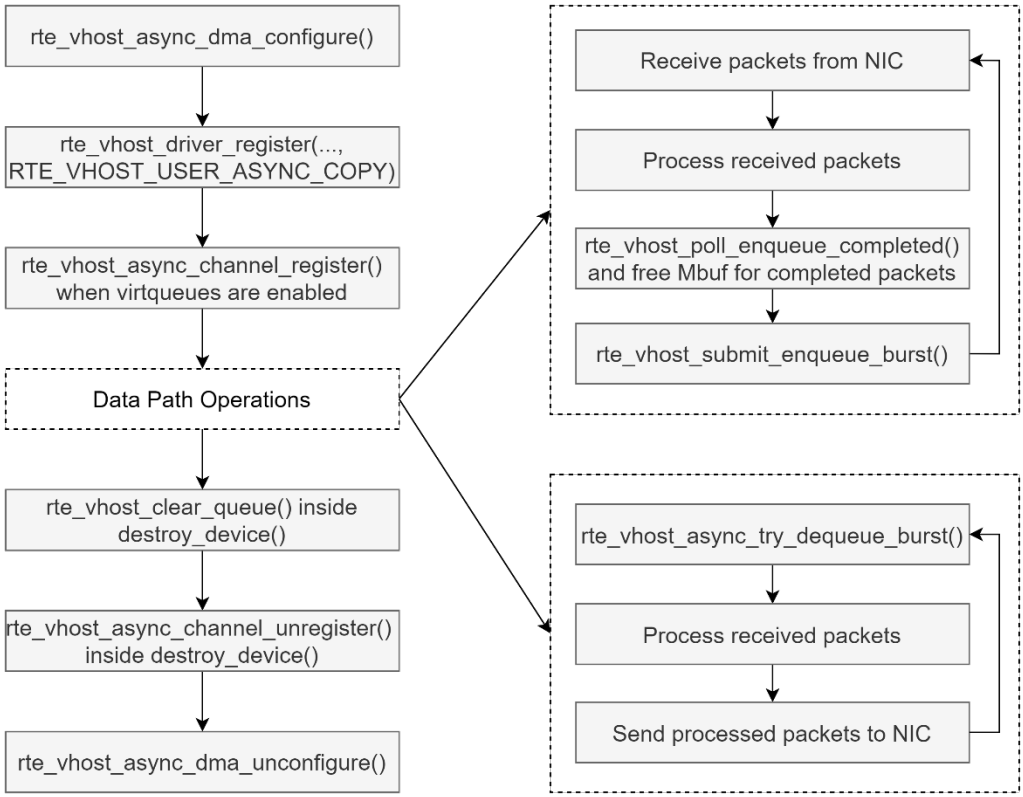

Figure 8. Example of Using DPDK Vhost Asynchronous APIs

3.4 Performance Pillars

3.4.1 Batching

Large batching size can reduce the average Intel DSA work submission cost. DPDK Vhost batches packet copies within one enqueue/dequeue operation and submits them to the Intel DSA via one Batch descriptor. Therefore, Vhost enqueue/dequeue burst size usually determines Intel DSA copy batching size.

Typically, DPDK packet burst size is 32. Thus, DPDK Vhost can batch 32 packet copies within one Batch descriptor at least, which is efficient for Intel DSA data copy. However, in some cases, the packet burst size per Vhost enqueue/dequeue may be much smaller, which hurts the Vhost performance, especially at small packet sizes.

3.4.2 Packet Size

Offloading packet copies to the Intel DSA can free core cycles for higher level functions, but at a cost of spending extra cycles on preparing descriptors, ringing doorbell to the Intel DSA and tracking asynchronous offloading status in Vhost. Therefore, for small packet sizes, like 64 bytes, software copying is more efficient. In this case, Intel DSA offloading cost will be higher than the gain, and using Intel DSA could result in performance decrease in turn.

3.4.3 Binding Intel® DSA WQs to Cores

Using Intel DSA WQs inside DPDK Vhost is protected with spinlock, as it is possible that multiple threads operate on the same Intel DSA WQ and the DPDK dmadev library does not support safe access to the Intel DSA WQ in the multi-core system. When multiple threads try to use the same Intel DSA WQ simultaneously, the Intel DSA WQ's spinlock will force the thread to busy waiting until acquiring the lock, causing wasting CPU time on useless activities. Binding Intel DSA WQs to cores can avoid contending for Intel DSA WQs among threads.

4 Performance Tests

We evaluate the performance of Host TestPMD with Intel DSA acceleration and compare it with the baseline Host TestPMD that is without Intel DSA acceleration. In all experiments, we assign one physical core to the host TestPMD. In the TestPMD with Intel DSA acceleration experiments, we use one Intel DSA instance. As this work uses DPDK Intel DSA driver that only supports the DWQ, we create two DWQs in the used Intel DSA instance. In addition, to use the maximum Intel DSA capability, all four engines in the Intel DSA instance are used.

4.1 Test Methodology

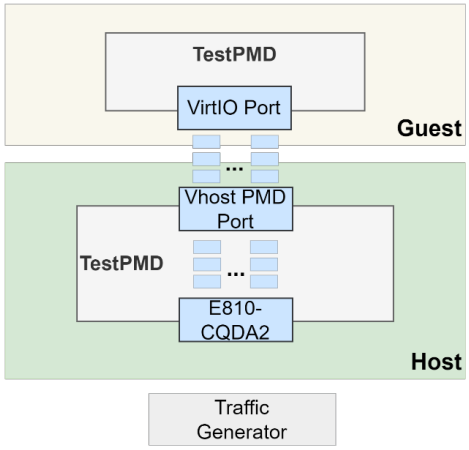

Figure 9. Experiment Topology

Figure 9 shows the experiment topology. In host, TestPMD runs in macfwd mode bound with one E810-CQDA2 port and one Vhost PMD port. The two DWQs are assigned to the TX queue and the RX queue in the Vhost PMD port respectively. In guest, TestPMD runs in macfwd mode bounded with a VirtIO port. The Ixia traffic generator connects to the E810-CQDA2 port and sends TCP/IPv4 packets with an acceptable loss rate of 0.1% for received traffic.

The hardware used in the experiments are listed in Table 3.

Table 3. Hardware Materials

CPU	Intel(R) Xeon(R) Platinum 8471N @ 1.80GHz 52 CPU cores * 1 NUMA nodes
Microcode	0x2b0000a1
Turbo	Off
Hyperthreading	Enabled
BIOS Version	EGSDCRB1.SYS.8901.P01.2209200243
Memory	32GB x 8 DIMMs x 1 NUMA nodes @ 4800MHz
Network Adapter	1x Intel® Ethernet Network Adapter E810-CQDA2 100Gb
Network Adapter Firmware	4.00 0x800117e9 1.3236.0

The software used in the experiments is listed in Table 4.

Table 4. Software Materials

Host Operating System	Ubuntu 22.04.1 LTS
QEMU	7.0.0
Host Kernel	5.15.0-27-generic
Host Compiler	GCC 11.2.0
Host DPDK	22.11-rc1 (a74b1b25136a), with applying three additional patches[9]
Guest Operating System	Ubuntu 22.04.1 LTS
Guest Compiler	GCC 11.2.0
Guest DPDK	22.11-rc1
Guest Kernel	5.15.0-27-generic

[9] The links of three patches: http://patches.dpdk.org/project/dpdk/patch/20221216020009.70206-1-yuanx.wang@intel.com/, http://patches.dpdk.org/project/dpdk/patch/20221011030803.16746-2-cheng1.jiang@intel.com/ and http://patches.dpdk.org/project/dpdk/patch/20221011030803.16746-3-cheng1.jiang@intel.com/.

4.2 Results

Figure 10 shows the packet forwarding rate of host TestPMD with and without Intel DSA acceleration. When the packet size is beyond 256 bytes, the packet forwarding rate of TestPMD with Intel DSA acceleration is 1.14-2.29 times as fast as the TestPMD without Intel DSA acceleration. In addition, TestPMD with Intel DSA acceleration achieves fixed packet forwarding rate from 64B to 1518B. With offloading packet copies to the Intel DSA, the host TestPMD only needs to operate the packet header, so the processing of TestPMD core is independent of packet sizes. In addition, the used Intel DSA bandwidth is less than the Intel DSA capability on data copy. Therefore, TestPMD with Intel DSA acceleration can achieve fixed packet forwarding rate crossing different packet sizes.

When packet size is smaller than or equal to 256 bytes, the performance of TestPMD with Intel DSA acceleration is worse. Since the core is more efficient than the Intel DSA on small copies and the Intel DSA offloading is expensive, Intel DSA offloading causes performance decrease for the host TestPMD.

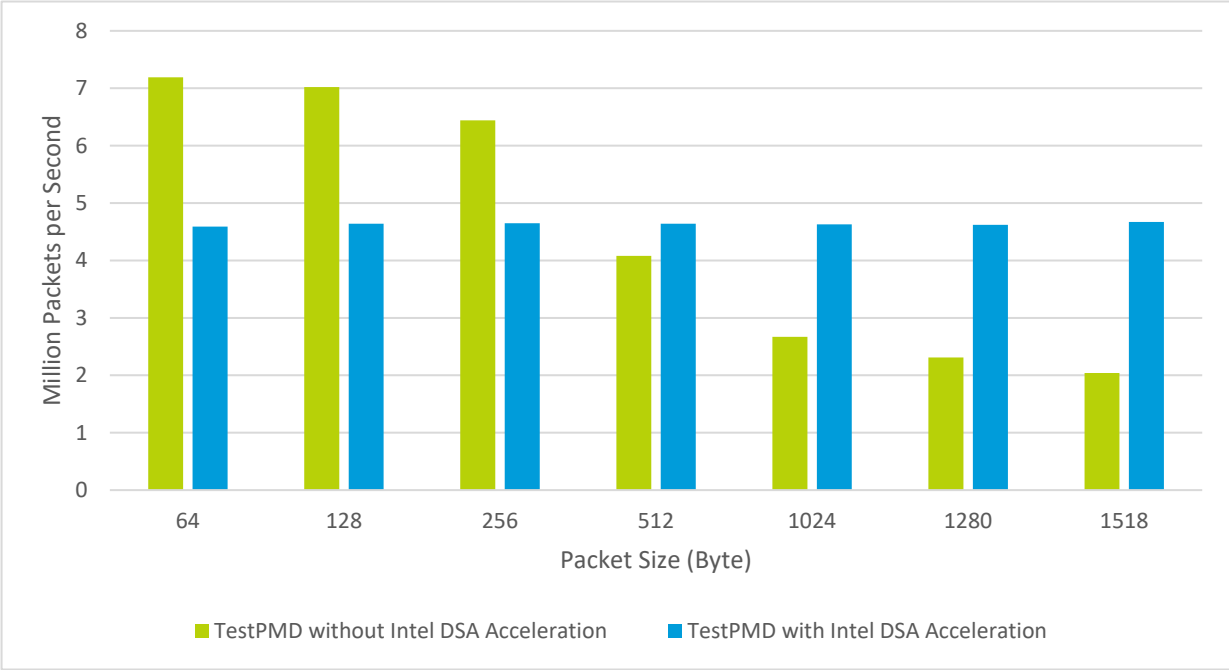

Figure 10. Packet Forwarding Rate Comparison

5 Summary

In this paper we have outlined the design of accelerating DPDK Vhost using Intel DSA and the usage of DPDK Vhost asynchronous APIs. We have benchmarked the packet forwarding rate with Intel DSA in 4th Gen Intel Xeon Scalable processors. Offloading packet copies to the Intel DSA brings considerable packet forwarding rate improvement for DPDK Vhost, when the packet size is beyond 256 bytes.

Appendix A Setup and Results

The Hardware and Software used in the experiments are in Table 3 and Table 4. Boot and BIOS settings are in Table 5.

Table 5. Boot and BIOS Settings

Item	Description
Host Boot Settings	GRUB_CMDLINE_LINUX="hugepagesz=1G hugepages=40 isolcpus=1-40,53-92 default_hugepagesz=1G rcu_nocbs=1-40,53-92 nohz_full=1-40,53-92 intel_pstate=disable processor.max_cstate=1 intel_idle.max_cstate=1 intel_iommu=on,sm_on iommu=on panic=30 init=/sbin/init net.ifnames=0 nmi_watchdog=0 audit=0 nosoftlockup hpet=disable mce=off tsc=reliable numa_balancing=disable memory_corruption_check=0 workqueue.power_efficient=false module_blacklist=ast modprobe.blacklist=ice init_on_alloc=0"
VM Boot Settings	hugepagesz=1G hugepages=4 default_hugepagesz=1G isolcpus=1-2 nohz_full=1-2 rcu_nocbs=1-2
BIOS	CPU C-state Disabled CPU P-state Disabled Turbo Disabled
Host Real Time Settings	echo -1 > /proc/sys/kernel/sched_rt_period_us echo -1 > /proc/sys/kernel/sched_rt_runtime_us echo 10 > /proc/sys/vm/stat_interval echo 0 > /proc/sys/kernel/watchdog_thresh
VM Real Time Settings	echo 0 > /proc/sys/kernel/watchdog echo 0 > /proc/sys/kernel/nmi_watchdog echo -1 > /proc/sys/kernel/sched_rt_period_us echo -1 > /proc/sys/kernel/sched_rt_runtime_us

Test steps for TestPMD without Intel DSA acceleration and Figure 3 are in Table 6.

Table 6. Test Steps for TestPMD without Intel® DSA Acceleration

Item	Description
Test Configuration	Test tool: IxNetwork 9.00.1915.16 Virtqueue size: 1024, the max size QEMU support Huge page size: 1GB VirtIO Mergeable: On Virtqueue Type: Split Virtqueue Forward Mode: TestPMD mac forward Vhost: 1 queue 1 logic core Virtio: 1 queue 1 logic core Totally 2 logic cores from 2 physical cores are used.
Flow Configuration	Ether()/IP()/TCP()
Test Step	1. Bind one E810-CQDA2 port to vfio-pci. 2. Mount 1GB huge pages for the host TestPMD and VM. mkdir /dev/hugepages mount -t hugetlbfs -o pagesize=1G nodev /dev/hugepages mkdir /mnt/huge mount -t hugetlbfs -o pagesize=1G nodev /mnt/huge

	3. Launch the host TestPMD and running in the macfwd mode. chrt -f 95 ./build/app/dpdk-testpmd -n 8 -l 9-10 --file-prefix=dpdk_vhost --huge-dir=/dev/hugepages --socket-mem 8192 --vdev 'net_vhost,iface=./vhost-net,queues=1' --iova=va -- -i --nb-cores=1 --txq=1 --rxq=1 --txd=2048 --rxd=2048 testpmd>set fwd mac testpmd>start 4. Launch QEMU. taskset -c 11,12,13 /usr/local/qemu-7.0.0/bin/qemu-system-x86_64 -name us-vhost-vm1 -cpu host -enable-kvm -m 8192 -object memory-backend-file,id=mem,size=8192M,mem-path=/mnt/huge,share=on -numa node,memdev=mem -mem-prealloc -smp cores=2,sockets=1 -drive file=/home/osimg/ubuntu22-04.img -chardev socket,id=char0,path=./vhost-net -monitor unix:/tmp/vm2_monitor.sock,server,nowait -device e1000,netdev=ntts1 -netdev user,id=ntts1,hostfwd=tcp:127.0.0.1:6002-:22 -netdev type=vhost-user,id=mynet1,chardev=char0,vhostforce -device virtio-net-pci,mac=52:54:00:00:00:01,netdev=mynet1,mrg_rxbuf=on,rx_queue_size=1024,tx_queue_size=1024,csum=off,guest_csum=off,host_tso4=off,guest_tso4=off,guest_ecn=off -vnc :10 -monitor stdio 5. Use QEMU monitor to bind vCPUs with physical processors on host machine. qemu monitor: info cpus #check pid taskset -cp 12 xxx #xxx is the pid number taskset -cp 13 xxx 6. Enter the VM and bind the VirtIO port to vfio-pci. modprobe vfio-pci echo 1 > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode 7. Launch TestPMD in the VM and run it in the macfwd mode. ./build/app/dpdk-testpmd -c 0x3 -n 8 -- -i --nb-cores=1 –txq=1 --rxq=1 --txd=2048 --rxd=2048 testpmd>set fwd mac testpmd>start

Test steps for TestPMD with Intel DSA acceleration are Table 7.

Table 7. Test Steps for TestPMD with Intel® DSA Acceleration

Item	Description
Test Configuration	Test tool: IxNetwork 9.00.1915.16 Virtqueue size: 1024, the max size QEMU support Huge page size: 1GB VirtIO Mergeable: On Virtqueue Type: Split Virtqueue Forward Mode: TestPMD mac forward Vhost: 1 queue 1 logic core Virtio: 1 queue 1 logic core Totally 2 logic cores from 2 physical cores are used.
Flow Configuration	Ether()/IP()/TCP()
Test Step	1. Bind one E810-C port to vfio-pci. 2. Mount 1GB hugepage for the host TestPMD and VM. mkdir /dev/hugepages mount -t hugetlbfs -o pagesize=1G nodev /dev/hugepages

mkdir /mnt/huge

mount -t hugetlbfs -o pagesize=1G nodev /mnt/huge

3. Bind one Intel DSA instance to the idxd driver, then create two Dedicated WQs.

   accel-config disable-device dsa0

   accel-config config-engine dsa0/engine0.0 --group-id=0

   accel-config config-engine dsa0/engine0.1 --group-id=0

   accel-config config-engine dsa0/engine0.2 --group-id=0

   accel-config config-engine dsa0/engine0.3 --group-id=0

   accel-config config-wq dsa0/wq0.0 --group-id=0 --mode=dedicated --priority=10 --wq-size=64 --type=user --name=dpdk_vhost

   accel-config config-wq dsa0/wq0.1 --group-id=0 --mode=dedicated --priority=10 --wq-size=64 --type=user --name=dpdk_vhost

   accel-config enable-device dsa0

   accel-config enable-wq dsa0/wq0.0

   accel-config enable-wq dsa0/wq0.1

4. Launch the host TestPMD and run it in the macfwd mode.

   chrt -f 95 ./build/app/dpdk-testpmd -n 8 -l 9-10 --file-prefix=dpdk_vhost --huge-dir=/dev/hugepages --socket-mem 8192 --vdev 'net_vhost,iface=./vhost-net,queues=1,dmas=[txq0@wq0.0;rxq0@wq0.1]' --iova=va -- -i --nb-cores=1 --txq=1 --rxq=1 --txd=2048 --rxd=2048

   testpmd>set fwd mac

   testpmd>start

5. Launch QEMU.

   taskset -c 11,12,13 /usr/local/qemu-7.0.0/bin/qemu-system-x86_64 -name us-vhost-vm1 -cpu host -enable-kvm -m 8192 -object memory-backend-file,id=mem,size=8192M,mem-path=/mnt/huge,share=on -numa node,memdev=mem -mem-prealloc -smp cores=2,sockets=1 -drive file=/home/osimg/ubuntu22-04.img -chardev socket,id=char0,path=./vhost-net -monitor unix:/tmp/vm2_monitor.sock,server,nowait -device e1000,netdev=ntts1 -netdev user,id=ntts1,hostfwd=tcp:127.0.0.1:6002-:22 -netdev type=vhost-user,id=mynet1,chardev=char0,vhostforce -device virtio-net-pci,mac=52:54:00:00:00:01,netdev=mynet1,mrg_rxbuf=on,rx_queue_size=1024,tx_queue_size=1024,csum=off,guest_csum=off,host_tso4=off,guest_tso4=off,guest_ecn=off -vnc :10 -monitor stdio

6. Use QEMU monitor to bind vCPUs with physical processors on host machine.

   qemu monitor: info cpus #check pid

   taskset -cp 12 xxx #xxx is the pid number

   taskset -cp 13 xxx

7. Enter the VM and bind the VirtIO port to vfio-pci.

   modprobe vfio-pci

   echo 1 > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode

8. Launch TestPMD in the VM and run it in the macfwd mode.

   ./build/app/dpdk-testpmd -c 0x3 -n 8 -- -i --nb-cores=1 --txq=1 --rxq=1 --txd=2048 --rxd=2048

   testpmd>set fwd mac

   testpmd>start

Test results for TestPMD with and without Intel DSA acceleration are in Table 8.

Table 8. Test Results for TestPMD with and without Intel® DSA Acceleration

	TestPMD without Intel DSA Acceleration (Million Packets per Second)	TestPMD with Intel DSA Acceleration (Million Packets per Second)
64	7.19	4.59
128	7.02	4.64
256	6.44	4.65
512	4.08	4.64
1024	2.67	4.63
1280	2.31	4.62
1518	2.04	4.67

1222/DN/WIT/PDF 758358-001US