



# Virtual Networking Solution and Performance on Arm Neoverse

110057\_0100\_00

Howard Zhang  
Yanqin Wei

This white paper introduces four mainstream virtual networking scenarios used in Kernel-based Virtual Machines (KVM), describes their implementation on an Arm64 server, and provides a comprehensive benchmark analysis of these network scenarios.

---

## 1 Overview

This paper introduces four mainstream virtual networking scenarios used in Kernel-based Virtual Machines (KVM). It describes their software and hardware implementation on the Arm64 server. Then it provides a comprehensive benchmark analysis of these network scenarios on the Arm Neoverse N2 server, highlighting their performance and efficiency.

The Arm architecture provides robust support for all these virtual networking technologies, making them easily deployable. All related software demonstrates strong compatibility and performance on the Arm Neoverse server. This ensures that virtualized networking solutions operate efficiently and stably on the Arm.

## 2 Background

Virtual networking is a software system operating on physical servers. It simulates a physical network to interconnect virtual machines. It facilitates effective communication within the same or across different servers. While physical networking relies on cabling and hardware, virtual networking implements these capabilities through software. This approach optimizes network flexibility and management. Figure 1 shows the main components of the virtual network. It is typically composed of virtual network devices and virtual switches. The virtual device has a frontend-backend architecture.

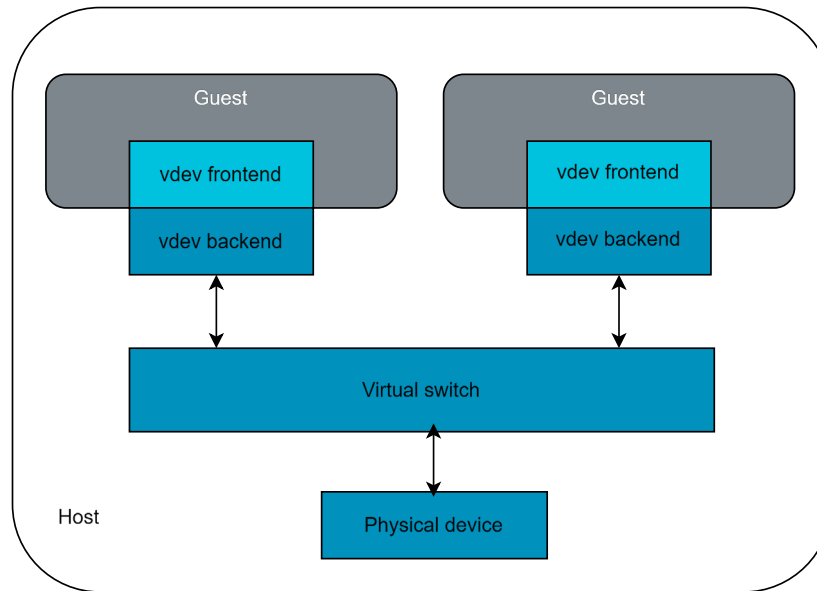


Figure 1: Virtual networking component

- **Frontend** is on the guest side. It presents as an emulated device. It is also known as the driver side.
- **Backend** is the peer of the frontend on the host side. It can be in a virtual machine monitor, such as Qemu, in a host kernel, or in a host userspace like Data Plane Development Kit (DPDK). The backend is also known as the device side.

Different virtual networks have varied implementations of the control plane and data plane. The control plane is responsible for negotiating communication protocols between the backend and frontend. The data plane handles the actual data transmission.

This paper focuses on the performance of the data plane, which includes:

- **Data Transport:** Data exchange between the backend and frontend via shared memory.
- **Notification:** Signal notification for synchronizing.

## 2.1 Virtual network solution

There are various virtual network solutions in the cloud computing area. This paper introduces four mainstream solutions including:

- Virtio-net device
- Vhost-net
- DPDK vhost-user
- VFIO-SRIOV

Arm has already established a comprehensive software ecosystem to support these solutions. They can run efficiently on Arm servers.

### 2.1.1 Virtio-net device

VirtIO[1] is a standardized interface that provides virtual machines with access to simplified virtual devices, such as block storage, network interfaces, and consoles. Data transmission of VirtIO is implemented by a specialized data structure known as virtqueues. It is a mechanism for bulk data transport on VirtIO devices.

- Frontend makes requests available to the backend by adding an available buffer to one virtqueue and optionally notifying an event to the backend.
- Backend executes the requests and adds a used buffer to another virtqueue. The backend can then trigger a notification event to the frontend.

QEMU implemented a virtio-net backend in the userspace called virtio-net device. Figure 2 shows the interaction flow in this case.

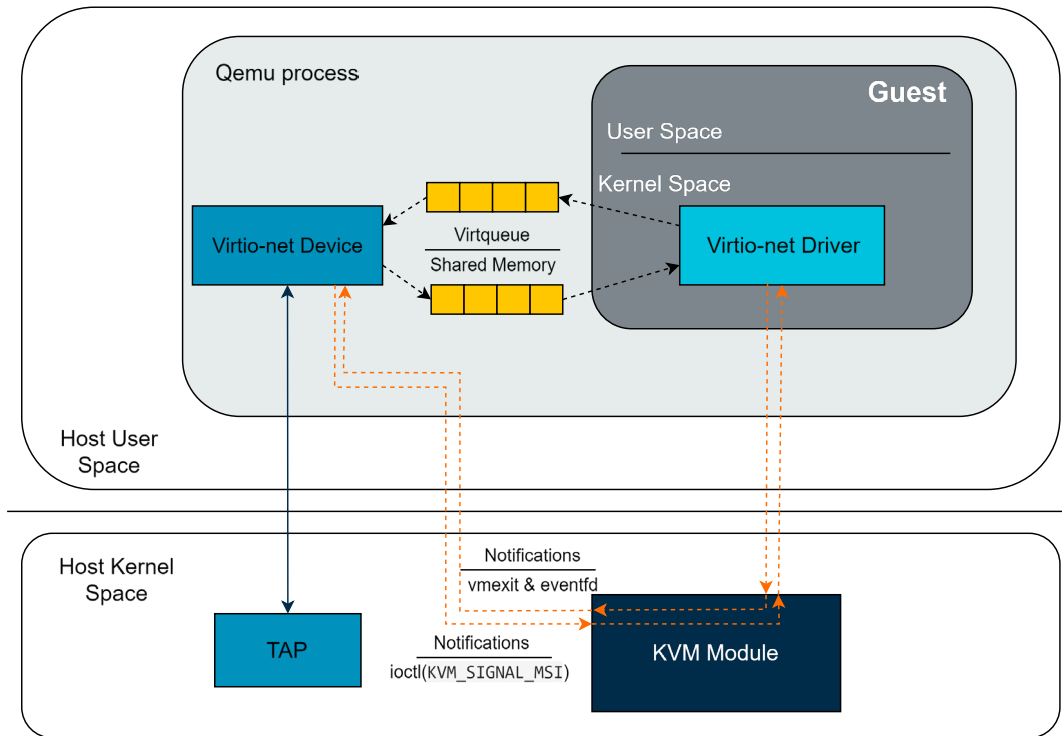


Figure 2: Qemu virtio-net device

Virtio-net backend sends/receives the networking data to/from the host kernel through the tap interface. The notifications mechanism is implemented by ioeventfd and KVM's ioctl.

#### Frontend to Backend:

- QEMU registers an ioeventfd for each device's Memory Mapped Input/Output (MMIO) Region.
- Frontend in the guest accesses this MMIO address to trigger a VM exit.
- KVM handles the VM exit and notifies the user-space QEMU.
- The QEMU main loop detects the event and transmits the packet to the tap interface.

#### Backend to Frontend:

- When a packet arrives, the corresponding tap device file descriptor becomes readable.

- The QEMU main loop receives an event and calls the callback function to handle the packet.
- Virtio-net device fills the packet data into the virtqueue.
- Virtio-net device uses KVM's ioctl to inject an interrupt, notifying the frontend driver to receive the packet.

### 2.1.2 Vhost-net

Because the virtio-net device operates in userspace, it must transfer data and notifications to the host kernel. This causes significant overhead. To mitigate the inefficiencies, the kernel vhost-net module was introduced. Figure 3 shows the interaction flow in this case.

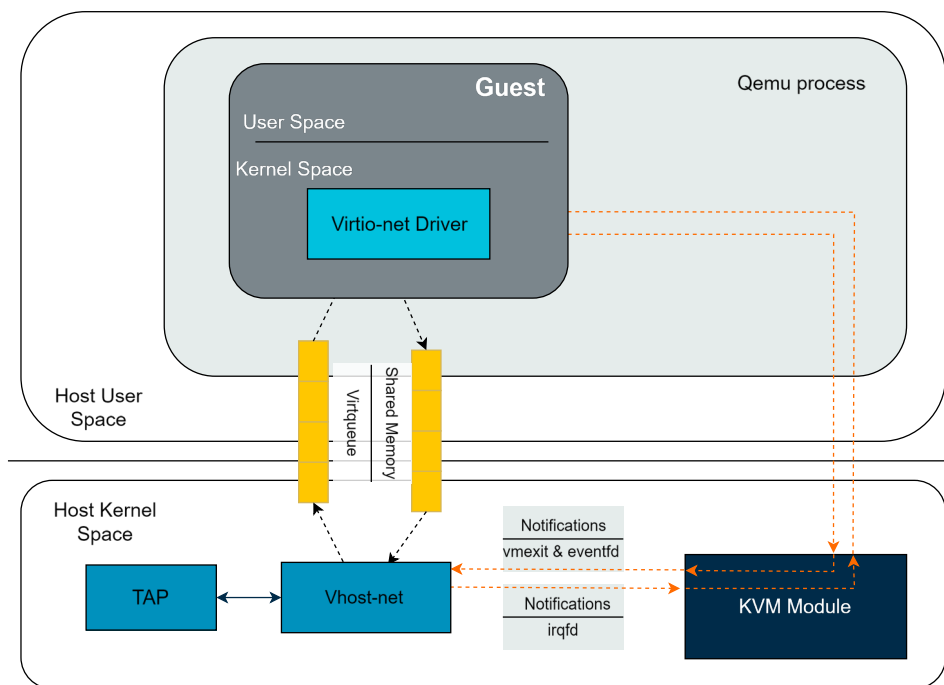


Figure 3: Linux kernel vhost-net

Vhost-net module in Linux provides in-kernel device emulation via vhost protocol. This protocol is a set of messages and mechanisms designed to offload the virtio datapath from QEMU to a kernel module. This enables device emulation code to directly call into kernel subsystems

instead of performing system calls from userspace[2]. The processing sequence of vhost-net solution is

Frontend to Backend:

- Qemu creates an ioeventfd for the guest hardware register.
- Guest puts packet into the queue, writes to the register, and then triggers a VM exit.
- KVM handles the VM exit and notifies vhost-net.
- The vhost-net polls this eventfd and transmits the packet to the tap interface.

Backend to Frontend:

- QEMU allocates an irqfd and registers it with both KVM and vhost-net.
- When a packet arrives, the vhost-net enqueues the packet and writes to the irqfd
- KVM polls irqfd and triggers an interrupt to the guest.
- The guest driver receives the notification to retrieve the packet.

Comparing the data flow between the virtio-net device and the vhost-net module, we can find:

- In the virtio-net device approach, data must be transferred between the QEMU userspace and the host kernel through system calls, this introduces additional overhead.
- Vhost-net allows data exchange between the tap device and vhost-net without address space copy and context switch.

Therefore, the vhost-net solution significantly improves the performance of virtual networking.

### 2.1.3 DPDK vhost-user

DPDK[3] is an open-source software library for high-performance packet processing. To enhance network performance, a DPDK application can be implemented entirely in user space, bypassing the kernel's networking stack.

DPDK has a vhost library that provides a high-performance VirtIO backend implementation for virtio-net devices, known as vhost-user. The virtio-net device emulation is offloaded to a DPDK-based application instead of using the kernel's vhost-net module. This is helpful for scenarios using user-space virtual switches, allowing data plane to bypass the kernel directly.

### 2.1.3.1 Control plane

The DPDK vhost library uses Unix domain sockets for vhost protocol communication between QEMU and DPDK. It supports both server and client modes. After establishing a connection, DPDK vhost-user processes vhost messages from QEMU. The frontend and backend negotiate features like memory region and vring configuration. They also exchange eventfd descriptors for event notifications.

### 2.1.3.2 Data plane

Figure 4 shows the dataplane of the DPDK vhost-user solution. It maps the guest's memory into the address space used by the DPDK application in the control plane. Therefore, the DPDK application can read/write packet buffers directly from/to guest memory.

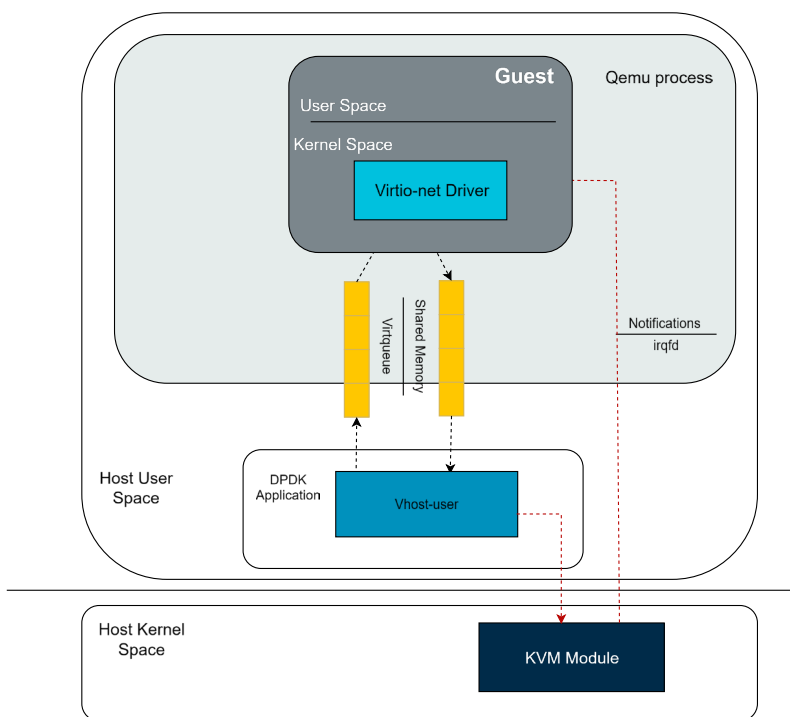


Figure 4: DPDK vhost-user dataplane

All brand names or product names are the property of their respective holders. Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder. The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given in good faith. All warranties implied or expressed, including but not limited to implied warranties of satisfactory quality or fitness for purpose are excluded. This document is intended only to provide information to the reader about the product. To the extent permitted by local laws Arm shall not be liable for any loss or damage arising from the use of any information in this document or any error or omission in such information.

This is the process by which a DPDK application receives packets from a physical NIC and forwards them to a guest VM:

- NIC to DPDK:

DPDK application continuously polls the physical NIC for incoming packets in userspace. It bypasses the kernel to ensure low latency.

- DPDK to Guest:

- DPDK vhost-user puts the packet into the shared queue.
- It writes the eventfd to notify that the packet buffer is available.
- KVM receives the event and injects an interrupt to the guest.
- The guest driver gets the notification to retrieve the packet.

The network packet transmission bypasses the host's kernel space except for the notification. This helps to improve performance and reduce network latency.

### 2.1.3 VFIO and SRIOV

VFIO[5] is a framework in the Linux kernel that enables userspace applications to directly access hardware devices without going through traditional kernel drivers. The guest can improve the networking performance by accessing the host network device directly. In the case of Qemu-KVM, a device can be assigned to a guest VM. It allows transparent communication with the device, including DMA memory mapping and interrupt handling. Figure 5 shows the use of VFIO for direct access to network devices that support single Root-I/O Virtualization (SR-IOV).



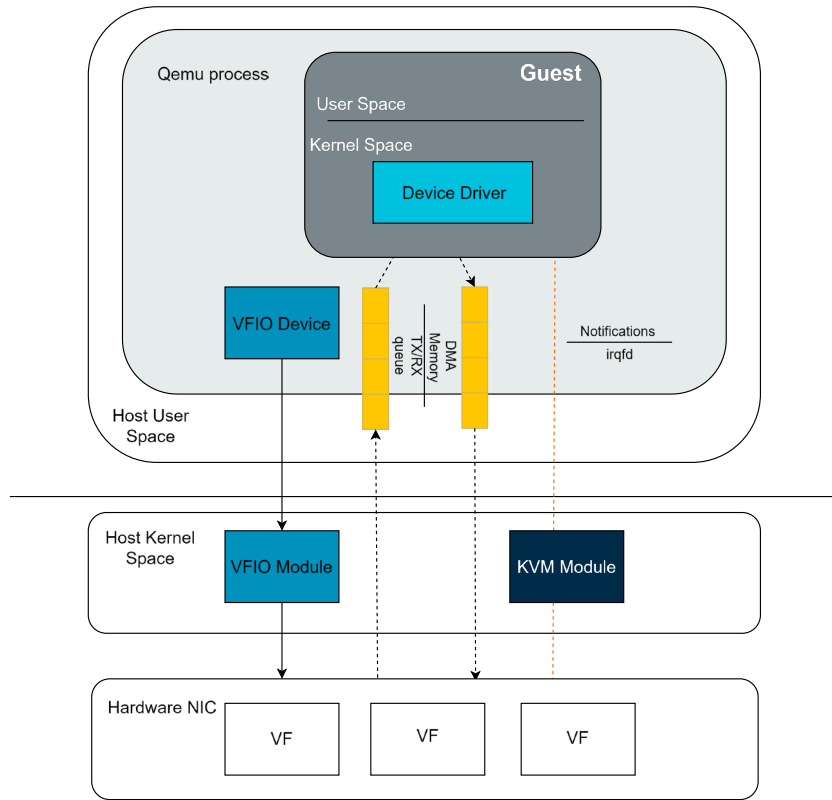


Figure 5: VFIO-SRIOV

SR-IOV can segment a compliant network device, recognized on the host node as a Physical Function (PF), into multiple Virtual Functions (VFs). The hardware supporting SRIOV can be exposed in several copies, called VF.

- The target device can be bound to the VFIO driver, transferring control of the device to the VMM such as QEMU.
- DMA memory and interrupts are mapped to Guest via VFIO module.
- On Arm, VFIO works with the Generic Interrupt Controller (GIC) to handle device interrupts and pass them to the VM. When the network packet is received by the host, the interrupt handler writes the `irqfd` to notify KVM. KVM can then translate and inject the virtual interrupt into the guest VM.

- In the GICv4, the ITS can translate and inject the interrupt (as a vLPI) into the virtual processor without software intervention. This feature can improve the performance of VFIO-SRIOV solution.

### 3 Performance testing

To verify that these virtual network solutions can run efficiently on the Arm Neoverse platform, we use Neoverse N2 servers to test various network performance metrics of these solutions.

#### 3.1 Testing topology

Figure 6 shows the topology of this performance test.

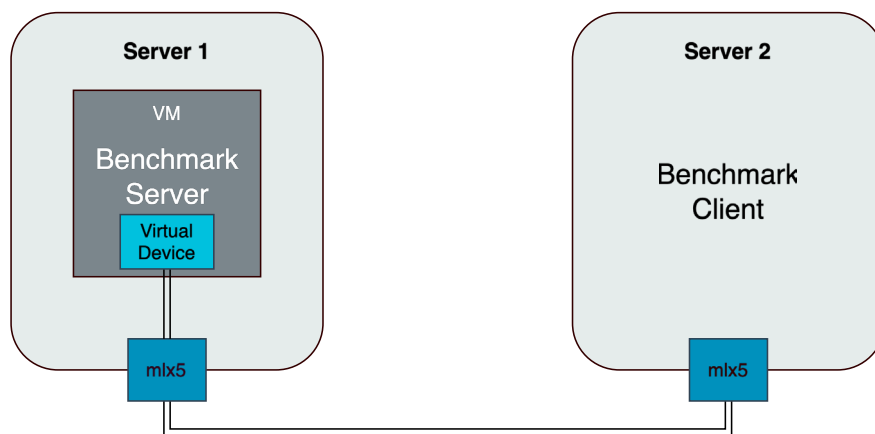


Figure 6: Testing topology

We conducted testing using two Arm Neoverse N2 servers, each equipped with a MLX5 40G network card. Figure 6 shows the topology of the testing. On Server 1, we launched a virtual machine and applied various virtual networking scenarios. Server 2 is the benchmark client to generate test traffic.

#### 3.1 Testing methodology

This section describes the testing methodology used in this paper. We use three kinds of measurements to evaluate network performance:

### 3.2.1 TCP network bandwidth

TCP network bandwidth is a crucial metric for evaluating network performance. It measures the maximum rate at which data can be transmitted over a TCP connection. It indicates the throughput and capacity between two endpoints.

#### 3.2.1.1 Test method

We use `iperf3` to measure bandwidth. The `iperf3` client sends data streams to the server using TCP to ensure reliable delivery. The test runs for 1 minute and records the average bandwidth.

#### 3.2.1.2 Test command

- Server: `taskset -c 1 iperf3 -s`
- Client: `iperf3 -c server_ip -t 60`

#### 3.2.1.3 Output description

The output should look like the following:

[ ID]	Interval		Transfer	Bitrate	Retr	
[ 5]	0.00-10.00	sec	30.4 GBytes	26.1 Gbits/sec	0	sender
[ 5]	0.00-10.04	sec	30.4 GBytes	26.0 Gbits/sec		receiver

We use the Bitrate for the receiver as the bandwidth. For the example above, the bandwidth is 26.0 Gbits/sec.

### 3.2.2 Packets Per Second (PPS)

PPS provides another view of the performance differences across various network scenarios. Since CPU usage for handling packet translation and interrupts varies, the efficiency for handling network packets also differs. PPS helps us understand this aspect.

#### 3.2.2.1 Test method

We use `iperf3` to measure PPS. The `iperf3` client sends UDP packets to the server. To fully utilize the CPU, we use multi-threaded `iperf3` to generate small-size UDP packets. It ensures the server's CPU usage reaches approximately 100%. However, packet drops are inevitable

when CPU usage reaches 100%. To maximize CPU usage and keep the packet drop rate low, we measure the largest PPS when the packet drop rate is below 0.1% in 1 minute.

### 3.2.2.2 Test command

- Server:  
`taskset -c 1 iperf3 -s`
- Client:  
`iperf3 -c server_ip -u -l 46 -b 120m -t 60 -P 4`
  - `-u` use UDP
  - `-l` length of buffer to read or write (small data length for maximum packets)
  - `-b` target bitrate in bits/sec (adjust to control packet rate)
  - `-t` time in seconds to transmit for
  - `-P` number of parallel client streams to run

### 3.2.2.3 Output description

The output looks like the following:

[ ID]	Interval		Transfer	Bitrate	Jitter	Lost/Total Datagrams	
[ 5]	0.00-60.04	sec	843 MBytes	118 Mbits/sec	0.003 ms	13900/19239054 (0.072%)	receiver
[ 6]	0.00-60.04	sec	843 MBytes	118 Mbits/sec	0.000 ms	15117/19238861 (0.079%)	receiver
[ 9]	0.00-60.04	sec	843 MBytes	118 Mbits/sec	0.001 ms	14790/19239031 (0.077%)	receiver
[ 11]	0.00-60.04	sec	843 MBytes	118 Mbits/sec	0.001 ms	15390/19238896 (0.08%)	receiver
[SUM]	0.00-60.04	sec	3.29 GBytes	471 Mbits/sec	0.001 ms	59197/76955842 (0.077%)	receiver

We use the `Lost/Total Datagrams` of the `SUM` row for calculation. The formula is:

$PPS = (\text{total packages} - \text{lost packages}) / 60 \text{ seconds}$

For the example above, the PPS is:

$(76955842 - 59197) / 60 = 1281610$

### 3.2.3 Network traffic delay

Network traffic delay measures how quickly a network scenario can respond to a request. Due to different notification and data translation paths, response speeds vary across network scenarios.

#### 3.2.3.1 Test method

We use `netperf` to test network traffic delays. The `TCP_RR` and `UDP_RR` (Request-Response) tests in `netperf` report round-trip latency for TCP and UDP traffic. We use `UDP_RR` and focus on mean latency and 99th percentile latency to evaluate network traffic delay.

#### 3.2.3.2 Test command

- Server:  
`taskset -c 1 netserver -D`
- Client:  
`netperf -H server_ip -D10,10 -t UDP_RR -l 60 -- -r 256 -o min_latency,mean_latency,p99_latency,max_latency`
  - `-D` Display interim results at least every time interval using units as the initial guess for units per second
  - `-t` Specify test to perform (`UDP_RR`)
  - `-l` Specify test duration
  - `-r` Set request/response sizes
  - `-o` Choose the output information

3.2.3.3 Output description

The output looks like this:

```
MIGRATED TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 192.168.0.3 () port 0
AF_INET : demo : first burst 0
43239.08,Trans/s,1.001,1716803280.976
43130.66,Trans/s,1.003,1716803281.978
43053.12,Trans/s,1.002,1716803282.980
44887.11,Trans/s,1.001,1716803283.981
44100.81,Trans/s,1.018,1716803284.999
42961.99,Trans/s,1.027,1716803286.025
43361.61,Trans/s,1.000,1716803287.026
42818.42,Trans/s,1.013,1716803288.038
43493.73,Trans/s,1.000,1716803289.039
43055.07,Trans/s,0.936,1716803289.975
Minimum Latency Microseconds, Mean Latency Microseconds,99th Percentile Latency Microseconds,
Maximum Latency Microseconds
19,22.98,27,488
```

We focus on the Mean Latency Microseconds and 99th Percentile Latency Microseconds. For the example above, the values are 22.98 and 27, respectively.

3.3 Testing setup

3.3.1 Software versions

Table 1 shows the software version used in the testing.

Software	Version	Source
Host Distro	ubuntu 22.04.4 LTS	-
Host Kernel	5.15.0-107-generic	-
Guest Distro	ubuntu 22.04.3 LTS	-
Guest Kernel	5.15.0-91-generic	-
Qemu	6.2.0	official deb package
DPDK	23.11	<a href="https://github.com/DPDK/dpdk">https://github.com/DPDK/dpdk</a>
OpenvSwitch	3.3.0	<a href="https://github.com/openvswitch/ovs">https://github.com/openvswitch/ovs</a>
Iperf3	Client: 3.16	<a href="https://github.com/esnet/iperf">https://github.com/esnet/iperf</a>
	Server: 3.9	official deb package
Netperf	2.7.0	official deb package

Table 1: Software Version



All brand names or product names are the property of their respective holders. Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder. The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given in good faith. All warranties implied or expressed, including but not limited to implied warranties of satisfactory quality or fitness for purpose are excluded. This document is intended only to provide information to the reader about the product. To the extent permitted by local laws Arm shall not be liable for any loss or damage arising from the use of any information in this document or any error or omission in such information.

## 3.3.2 Virtual machine setup

### 3.3.2.1 Virtio-net and vhost-net

We use the Qemu command to start up the VM directly.

The parameter difference in virtio-net device and vhost-net scenarios is "-netdev". The vhost is disabled in virtio-net device case and enabled in vhost-net case.

General virtual machine command:

```
qemu-system-aarch64 \
-enable-kvm \
-m 8192 \
-cpu host \
-smp 2 \
-M virt \
-nographic \
-kernel /vmlinuz-5.15.0-91-generic --append "root=LABEL=cloudimg-rootfs ro nosplash
console=ttyAMA0 console=tty1 console=ttyS0" \
-initrd /initrd.img-5.15.0-91-generic \
-drive if=none,file=/jammy-server-cloudimg-arm64.img,id=hd0 \
-device virtio-blk-pci,drive=hd0 \
```

virtio-net device:

```
-netdev tap,id=tap0,script=qemu-ifup,vhost=off \
-device virtio-net-pci,netdev=tap0
```

vhost-net:

```
-netdev tap,id=tap0,script=qemu-ifup,vhost=on \
-device virtio-net-pci,netdev=tap0
```

### 3.3.2.2 DPDK vhost-user and Open vSwitch

Open vSwitch supports a DPDK usespace datapath. It can be used to test the performance of vhost-user. In this test case, we need to start up the OVS-DPDK application.

```
# start server
$ /usr/local/bin/ovsdb-tool create /usr/local/etc/openvswitch/conf.db
/usr/local/share/openvswitch/vswitch.ovsschema
$ /usr/local/sbin/ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock --
remote=db:Open_vSwitch,Open_vSwitch,manager_options --pidfile -detach

# start daemon
$ /usr/local/sbin/ovs-vswitchd unix:/usr/local/var/run/openvswitch/db.sock --pidfile --detach --
log-file=/usr/local/var/log/openvswitch/ovs-vswitchd.log
```

```
# use openvswitch to create dpdk bridge, and bind dpdk to the bridge
sudo ovs-vsctl set Open_vSwitch . other_config:dpdk-init=true
sudo ovs-vsctl set Open_vSwitch . other_config:pmd-cpu-mask=0xff0000
sudo ovs-vsctl add-br dpdk-br0 -- set bridge dpdk-br0 datapath_type=netdev

# check the NIC device ID by lspci -vnn, e.g. 0002:01:01.1
sudo ovs-vsctl add-port dpdk-br0 dpdk0 -- set Interface dpdk0 type=dpdk options:dpdk-
devargs=0002:01:00.1 ofport_request=1
sudo ovs-vsctl add-port dpdk-br0 vhost-user0 -- set Interface vhost-user0 type=dpdkvhostuser
ofport_request=2

#open flow forwarding rule
sudo ovs-ofctl -O OpenFlow11 add-flow dpdk-br0 in_port=2,action=output:1
sudo ovs-ofctl -O OpenFlow11 add-flow dpdk-br0 in_port=1,action=output:2
```

The distinguished parameters in Qemu command are:

```
-numa node,memdev=mem -mem-prealloc \
-object memory-backend-file,id=mem,size=2048M,mem-path=/dev/hugepages,share=on \
-chardev socket,id=char0,path=/usr/local/var/run/openvswitch/vhost-user0 \
-netdev type=vhost-user,id=vhostusr0,chardev=char0,vhostforce=on \
-device virtio-net-
pci,netdev=vhostusr0,mac=52:55:00:12:34:00,mrg_rxbuf=on,rx_queue_size=1024,tx_queue_size=1024
```

### 3.3.2.3 VFIO-SRIOV

We need to set up the virtual function interface for the vfio, then set up vfio-pci in Qemu process.

```
-device vfio-pci,host=0002:01:01.2
```

### 3.3.3 Miscellaneous configuration

To avoid any distractions from memory access, interrupts, and task scheduling, we do the following:

- Soft-offline all CPU other than numa node 0.
- Stop irqbalance.
- Bind the network card irq with cpu core.
- Bind vcpu with cpu.
- Bind iperf process with vcpu.
- Set kernel parameter to enable iommu passthrough.



- Set the CPU into performance mode.

### 3.3.3.1 Configurations

- NUMA Setup and CPU Pinning

To ensure consistent performance and minimize scheduling overhead, we only activated the CPU cores on NUMA node 0 and deactivated the others. Additionally, we pinned the vCPU, QEMU process, and virtual networking-related processes to specific cores on NUMA node 0. Table 2 show the CPU affinity of the processes and threads.

Network scenario	Pinned processes/threads
virtio-net device	Qemu process, vCPU threads
vhost-net	Qemu process, vCPU threads, vhost-net process
DPDK vhost-user	Qemu process, vCPU threads, DPDK eth pmd thread, DPDK vhost-user pmd thread
VFIO-SRIOV	Qemu process, vCPU threads

Table 2: CPU Affinity of Threads

- Socket Buffer Configuration

Given the high-speed network cards (40 Gbit/s), we set the socket buffer size to 25 MB. A larger socket buffer is necessary to handle the increased data throughput efficiently.

- Enable Virtio Device Rx Mergeable Buffer

Rx mergeable buffer is a virtio feature that allows the chaining of multiple virtio descriptors to handle large packet sizes. This feature is widely used in many user scenarios to allow the jumbo frame to be processed correctly.

# 4 Benchmark results

## 4.1 Network bandwidth benchmark

Figure 7 shows the bandwidth testing result.

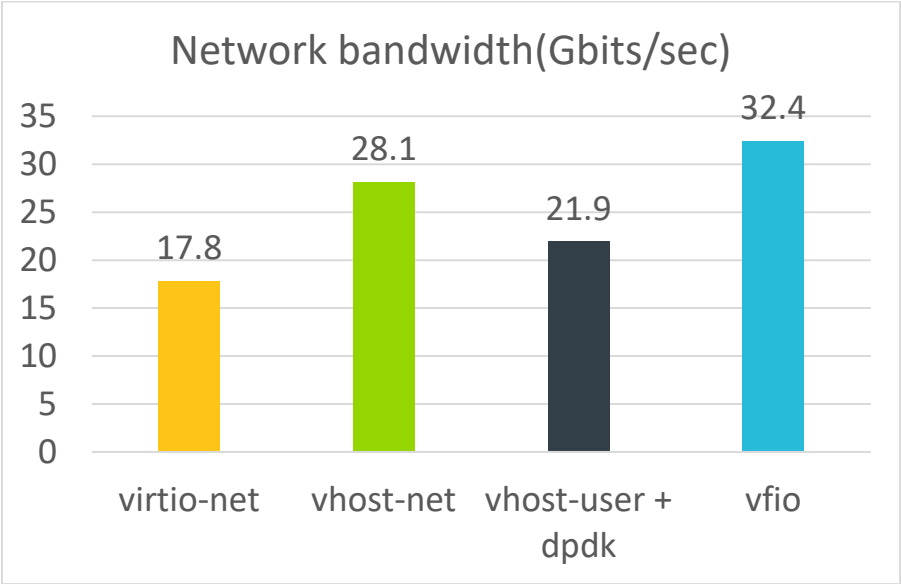


Figure 7: Bandwidth testing result

## 4.2 Packets Per Second benchmark

Figure 8 shows the PPS testing result.

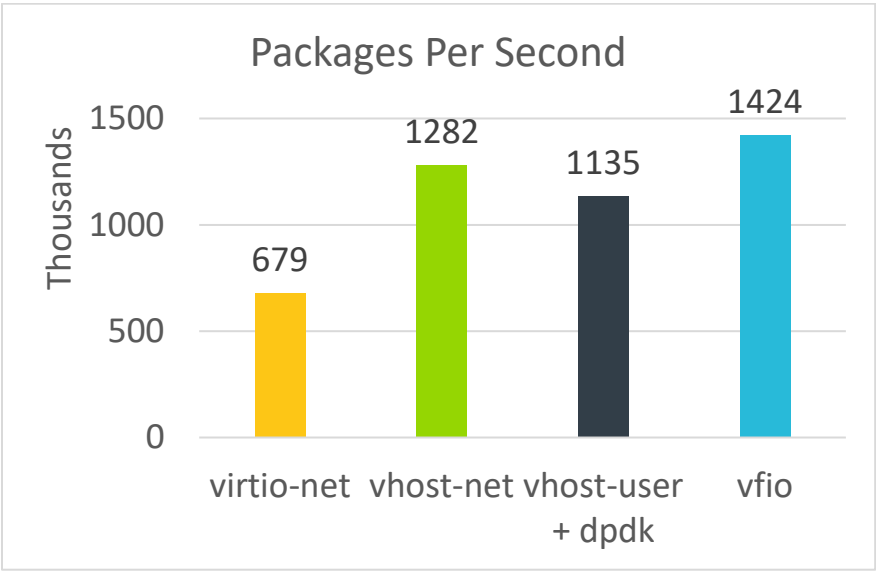
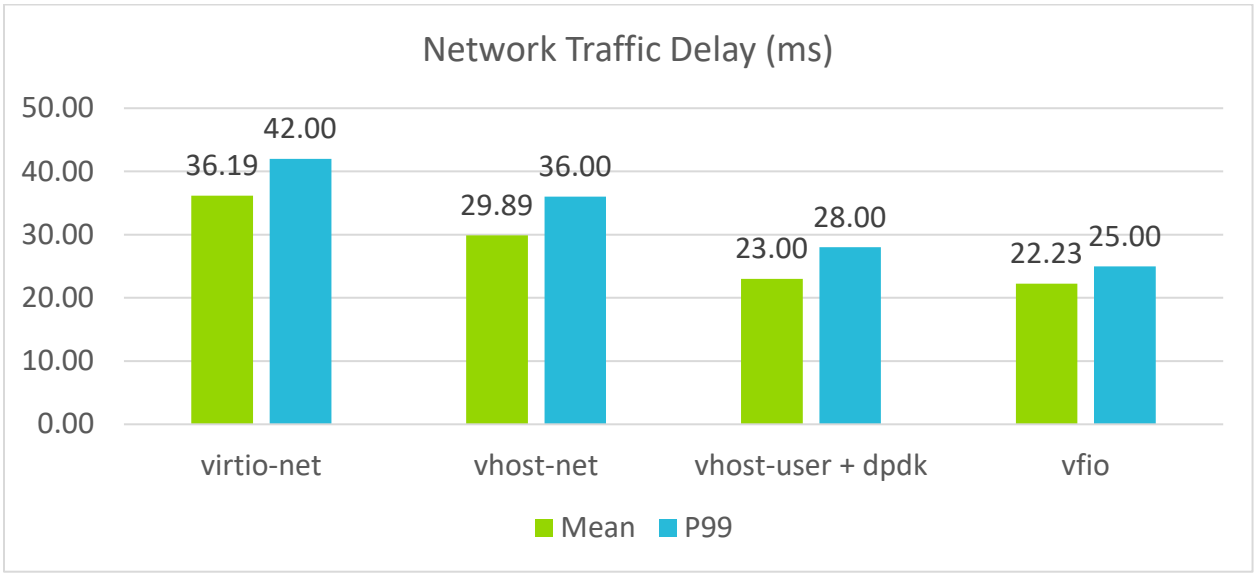


Figure 8: PPS testing result

## 4.3 Network traffic delay benchmark

Figure 9 shows the networking traffic delay testing result.



All brand names or product names are the property of their respective holders. Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder. The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given in good faith. All warranties implied or expressed, including but not limited to implied warranties of satisfactory quality or fitness for purpose are excluded. This document is intended only to provide information to the reader about the product. To the extent permitted by local laws Arm shall not be liable for any loss or damage arising from the use of any information in this document or any error or omission in such information.

Figure 9: Network traffic delay testing result

## 5 Benchmark Results Analysis

### 5.1 Virtio-net device

Across all benchmark tests, the virtio-net device has the worst performance. This is because this approach has a much longer data path and a more complex notification mechanism compared to other network scenarios. This longer process involves several steps:

- 1 The host kernel receives the packets from the physical network interface.
- 2 These packets are transferred to QEMU process in the host userspace through the tap interface.
- 3 The packets are delivered to the guest kernel by the virtqueue.

Network packets are passed through multiple layers, including the tap interface, QEMU process, and finally the guest kernel. This long path increases the overhead of context switch and memory copy between kernel and userspace.

The notification mechanism in virtio-net is also a performance bottleneck. The virtio-net device in userspace notifies the KVM by the ioctl system call, resulting in frequent context switches. These operations lead to degraded performance.

Given these complexities in both data and notification paths, the low performance of virtio-net is understandable and expected.

### 5.2 Vhost-net

On both network bandwidth and PPS benchmark, the kernel vhost-net scenario ranks second. Compared with the virtio-net device scenario, the vhost-net scenario has a shorter data path and a more efficient notification mechanism.

- Datapath

This approach moves the virtio backend data path from userspace (Qemu) to the kernel

space. By processing packets directly in the kernel, vhost-net eliminates the context switch and bulk data copies between userspace and kernel space.

- Notification
  - When data is ready to send, the guest writes MMIO region to trigger an ioeventfd signal to vhost-net kernel module.
  - When the data is ready to receive, the vhost-net writes irqfd to notify KVM, which signals an interrupt directly to the guest.

This efficient handling reduces the overhead from ioctl syscall and benefits overall performance.

### 5.3 DPDK vhost-user

In throughput tests, including TCP bandwidth and UDP PPS, DPDK ranks third, slightly behind vhost-net. However, in network latency tests, it performs significantly better than vhost-net, ranking second.

DPDK uses userspace polling threads to process network traffic. These threads run on dedicated CPU cores to avoid the overhead of context switching and interrupt handling. Additionally, it receives and forwards the packets in the userspace and bypasses the host kernel. It reduces the overhead in the kernel network stack.

However, in this test case, the performance bottleneck is the guest VM, not the DPDK application. This is because the PMD thread utilization observed is less than 50 percent.

#### 5.3.1 Throughput

Compared with vhost-net scenario, DPDK uses "run-to-completion model" for packet processing, which has a negative performance in this test case.

The PMD thread continuously polls for receiving packets from the physical NIC. It processes and forwards them to the guest in the same loop. Due to high polling frequency, only a small number of packets are handled in each loop. Each time packets are sent to the guest through

the vhost-user API, it triggers one virtual interrupt. This leads to more virtual interrupts and VM exits compared to vhost-net.

Use the following command to collect the number of VM exit events. Compare the number of events generated by receiving 1,000,000 network packets in vhost-user and vhost-net cases.

```
perf kvm stat record -p <pid>
```

Figure 10 shows that much more VM exit happens in the DPDK vhost-user test cases. This affects networking throughput results. In real-world deployment, the vhost-user backend can be used in conjunction with the DPDK virtio-net frontend. In this case, the DPDK frontend receives packets by polling, avoiding the overhead caused by numerous interrupts. However, this DPDK frontend is incompatible with the iperf3 tool in the guest VM, as it does not support DPDK APIs.

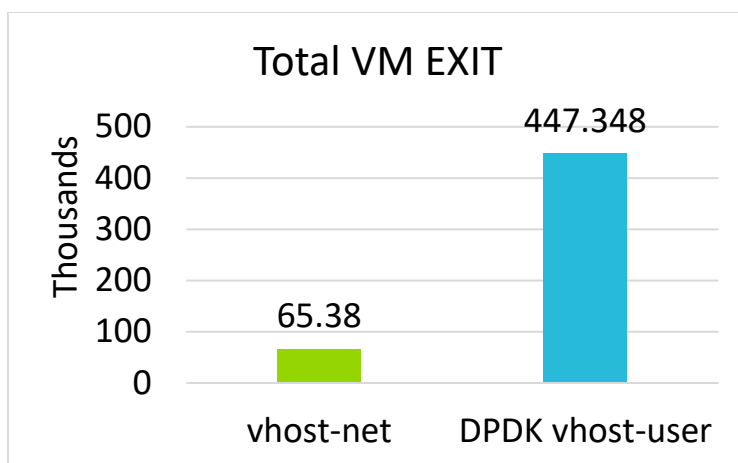


Figure 10: VM-Exit statistic

### 5.3.2 Latency

In latency tests, the DPDK vhost-user behaves better than vhost-net. This is also due to the polling and run-to-completion model. The polling mode avoids the overhead associated with interrupt handling during packet reception, and the run-to-completion model processes and transmits packets without task switching. It avoids the impact of task scheduling. Additionally, vhost-user exclusively utilizes the entire CPU core, eliminating context-switching overhead. Therefore, this approach is advantageous in latency-sensitive scenarios.

### 5.3 VFIO-SRIOV

In the VFIO-SRIOV solution, the virtual network backend is fully offloaded to the NIC hardware. The NIC provides virtual functions that allows the guest access network devices directly. Test results show that this solution offers the best performance in terms of both throughput and latency.

## 6 Conclusion

This paper analyzes four virtual network solutions in cloud computing on Arm Neoverse N2 servers, including virtio-net device, vhost-net, DPDK vhost-user, and VFIO-SRIOV. We aimed to evaluate their performance across three key metrics: TCP network bandwidth, Packets Per Second (PPS), and network traffic delay.

The benchmark results show that while virtio-net device offers a basic and flexible solution. Other scenarios are better because of its inefficient data path. Vhost-net improves it by offloading datapath to kernel space, resulting in better bandwidth and PPS performance. DPDK vhost-user further enhances performance by dedicating CPU cores to polling, significantly reducing latency and increasing efficiency. VFIO-SRIOV, though hardware-dependent, offers the highest throughput and lowest latency by leveraging NIC hardware capabilities.

All the test cases run smoothly on the Arm server. This demonstrates strong support for all relevant software and virtual networking technologies on the Arm Neoverse platform.

## 7 References

- [1] "Virtual I/O Device (VIRTIO) Version 1.2"  
<https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html>
- [2] "Deep dive into Virtio-networking and vhost-net"  
<https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net>
- [3] "DPDK vHost User Ports"  
<https://docs.openvswitch.org/en/latest/topics/dpdk/vhost-user/>
- [4] "linux kernel v5.15"  
<https://github.com/torvalds/linux/tree/v5.15>
- [5] "VFIO - 'Virtual Function I/O'"  
<https://docs.kernel.org/driver-api/vfio.html>
- [6] "Learn the architecture - Generic Interrupt Controller v3 and v4, Virtualization"  
<https://developer.arm.com/documentation/107627/0102/Virtualization>