# Journey of advancing device migration for virtio PCI hardware devices

**Parav Pandit** parav@nvidia.com, **Satananda Burla** sburla@marvell.com, **Yishai Hadas** yishaih@nvidia.com, **Avihai Horon** avihaih@nvidia.com, **Feng Liu** feliu@nvidia.com

**Abstract**

Virtio network devices are fundamental building blocks of cloud VMs. They have evolved from para-virtual devices to vDPA to hardware PCI devices, such as PCI physical and virtual functions. These PCI virtual functions are commonly attached as pass-through devices using the VFIO subsystem to VMs. Live migration of a VM with pass-through virtual functions is necessity of hypervisor infrastructure. This talk presents our journey, covering various design aspects, implementation challenges, their possible solutions, and performance benchmark results.

In this talk we highlight certain design considerations which steered the specification draft and its implementation. We also compare key differences of our approach with a vdpa vendor-based approach and briefly with the IDPF. This technical paper presents the live migration performance metrics for the first time globally for virtio PCI devices with IOMMU based dirty page tracking. We also discuss performance benchmark results from one to multiple devices and how the pre-copy approach further reduces migration downtime. Finally, we discuss the current progress of the Virtio specification developed by the open standards OASIS community. We conclude this talk by sharing our lessons learned from developing code and specifications in tandem, closing the gaps and discussing the trade-offs of various approaches.

**Keywords**

Virtio, live migration, checkpoint, snapshot, passthrough device, SR-IOV, virtual functions, vfio, IOMMU dirty page tracking, iommu fd, SIOV.

## 1. Introduction

Virtio network devices [1] are fundamental building blocks of a cloud VM. They have evolved from para-virtual devices to vDPA [2] to hardware PCI devices, such as PCI physical functions and PCI virtual functions [3]. These PCI virtual functions (VFs) are commonly attached as pass-through devices using the VFIO drivers [4] to a VM. Live migration of a VM with pass-through virtual functions is necessity of hypervisor infrastructure. However, passthrough hardware based virtual functions possess a challenge to migrate the VMs because such hardware device state is no longer part of a VM's memory. To migrate such a VM, the PCI device must have infrastructure accessible to the hypervisor so that source hypervisor can read the PCI device state and setup the same on the destination hypervisor. Our design covers the virtio specification extensions to support device migration functionality for the PCI VF devices. We discuss the design principles, specification proposal key elements, and its performance analysis using Nvidia bluefield-3 DPUs.

The requirements and design principles are described in section 2 which influences the migration interface design. We propose a migration interface using a PCI PF to handle the migration of the virtio device state of a PCI VFs. This migration interface can read or write the device state. Reading the device state is typically done after the PCI VF is stopped by source hypervisor. Device state is written when the PCI VF is stopped by the destination hypervisor. A PCI PF can stop or resume the PCI VF on an instruction from the migration stack. We also propose an open standard device state definition which consist of common and device type specific attributes which is migrated from the source to destination hypervisor and finally written to the device before resuming the device. This design is further described in section 3.

Section 4 compares virtio PCI device migration of our design with vendor specific vdpa approach using ConnectX6-DX device. In this section we show that though vdpa is useful for certain use cases, virtio PCI device based live migration has far less software components, lower number of operations during the VM downtime and eliminates traffic disruptive operation on the source hypervisor. This design results a better user experience and lower VM downtime. Section 5 brings a short comparison with the IDPF migration paper [25] presented in the netdev0x17.

Section 6 focuses on describing the various performance test results for measuring the VM downtime for various vCPU, memory, and device scale per VM configuration. We run performance tests with and without device pre-copy methods. We showcase that enabling device pre-copy mode that copies the device state while the VM is running delivers 494% reduction in the VM downtime for large scale VM and 340% reduction at small scale of 1 device per VM.

We finally summarize in section 7 the current state of drafted specifications, next steps of open-source drivers, certain improvement areas and future direction and lessons learned in the project. Section 8 concludes the discussion summarizing the modular and reusable building blocks, design simplicity and performance gain.

## 2. Design principles

The device migration interface design is based on four fundamental design requirements:

a. Achieve the lowest guest VM downtime.
b. Attain scalability, performance, and lower power consumption of the migration interface.
c. virtio-net features and other device extensions to have minimal changes to the migration interface.
d. Cloud operator to have single device migration framework covering multiple device types of NICs, GPUs, storage and other passthrough devices.
e. Lay the groundwork for supporting PCIe IDE, TDISP protocol, multi-PF NICs [5] to the guest VMs.

We considered the following design principles when defining the device migration interface between the driver and the device to address the above requirements:

1. Near O(1) downtime regardless of the number of devices in the guest VM. A guest VM may have one or more virtio devices of various types, each with numerous resources. For a virtio-net device, these resources include virtio transmit and receive queues, various queue attributes including their addresses, interrupt moderation attributes, RSS configuration, VLAN filter table, MAC filter table, and MAC address configuration. Unlike paravirtual software-based devices, where every configuration is nearly a memory access, in a real PCI hardware-based device, resource query and configuration are often a slow compared to memory access. This is because it involves programming one or more hardware ASIC modules, which include logic gates, TCAM, address translation modules, cache updating, finding optimal resources on a chip, and isolating these resources by setting up necessary access control guards. As the virtio-net device evolves, it is likely to have more device resources directly accessible from the guest VM, such as flow filters as proposed in [6]. Reading and writing more resources using a migration interface may result in an increased latency from the device of the migration commands. Therefore, a migration interface is needed that can amortize the high latency of accessing the device during the migration phase compared to nanoseconds level memory accesses from the hypervisor CPU's address space. Such an interface will have a direct effect on reducing VM downtime.
2. The migration interface should perform the minimal number of operations once the device is stopped on the source hypervisor, and a similarly minimal number of operations on the destination hypervisor while resuming the device. VM downtime is directly proportional to the number of operations during the time window when the VM is stopped on the source hypervisor and resumed on the destination hypervisor.
3. It is cited that SR-IOV VFs may reach a scalability limit beyond several thousand VFs due to the burden of on-chip configuration space limits, the MSI-X table accessible in the CPU address space, and the symmetrical configuration of the VF. To overcome this limit, scalable IO virtualization (SIOV) is being developed by the industry [7]. A migration interface should have only minimal differences when migrating either an SR-IOV virtio VF or a SIOV virtio function.
4. Although our research and experiments largely focused on the virtio-net device, we recognize that a guest VM often hosts various types of virtio devices in a cloud deployment, including virtio blk and virtio file system devices. A migration interface should be capable of migrating such non-virtio-net devices as well without requiring any additional changes.
5. A guest VM often deploys multiple PCI devices with different capabilities in a cloud environment. For example, a VM may include two virtio-net devices, each facing different networks in the data center: one connected to the data center's internal network, known as east-west communication, and the other connected to the external network, known as north-south communication. Alternatively, a guest VM may host multiple non-virtio devices such as RDMA [8], GPUs, or NVMe devices [9] as passthrough devices. It is crucial for the cloud system to utilize live migration stack across all device classes, thereby leveraging a common software stack in the hypervisor for all different device types, instead of creating and using a custom migration framework for each device type.
6. The addition of features to virtio-net devices has been on the rise in recent years. When a virtio-net VF device introduces a new feature, it is vital not to require upgrades to the migration stack to support migrating a device with this new feature. Such an interface results in a simpler device and migration driver design. Failure to adhere to this principle leads to software maintenance challenges, compatibility issues, and constant upgrades of the migration stack.
7. A guest VM may perform a virtio-level device reset while the migration interface needs to access the device, unaware of any ongoing migration as it must be transparent to the guest VM. Therefore, any migration interface should be designed such that a virtio device reset does not disrupt it. Similarly, a guest VM may perform a PCI VF level function level reset (FLR). Such PCI level operations must not affect the migration either. Additionally, an advanced VF may support PCI power management capabilities, allowing the VF to stop or resume the device directly. To support

such a scenario, the migration interface must not be halted when the VF is suspended. Furthermore, having a migration interface replicated on each of the VF that needs to be functional during the suspended state of the VF could exceed the PCI power budget envelope.

8. We anticipate that critical cloud applications in the guest VM would prefer to run without exposing data to the hypervisor system. The migration interface should not compromise the PCIe TDISP state machine to achieve confidential computing use cases and continue to support PCIe IDE. Therefore, the design of the migration interface must be independent of the VF itself, as a VF in the CONFIG_LOCKED state would not be directly accessible to the migration interface in the hypervisor.
9. A guest VM may have other devices that directly access the virtio-net device by performing driver notifications, also sometimes known as PCIe peer-to-peer communication [10]. It is essential for the migration interface to support such notifications while stopping the VM, because stopping a VM with multiple passthrough devices requires halting several devices simultaneously, which is not an atomic operation.
10. Only a handful of CPU/IOMMU platforms [11], [12] are capable of tracking memory writes performed by the IO device, often known as dirty page tracking. Virtio-net VF devices are used across many generations of CPUs, some of which may lack the ability to track pages written by devices. Therefore, migration interface to have optional capability to perform dirty page tracking for memory written by the device.

## 3. Design

Following these design principles, we developed straightforward virtio device interface requirements described in the following section:

a. An asynchronous queuing interface between the driver and the device that can issue migration commands to handle migrating devices.
b. Capability to issue one or more outstanding commands through this queuing interface.
c. Given that it must handle virtio device reset and PCI VF FLR without any mediation, the asynchronous interface should be separate from the PCI VF devices and PCI SIOV devices.
d. The interface should be able to suspend and resume the PCI VF.
e. The interface should be able to read and write the device state that can be migrated from source to destination via the DMA interface.
f. A device state structure that can be extended for new features and functionality, which the hypervisor driver can also decode and encode as necessary. This structure allows different cloud providers to program their virtio devices so that they are identical on both the source and destination systems for the PCI VF.
g. A device state structure that accommodates common fields across different types of virtio devices and avoids duplication of these fields across different device types.

It turned out that these requirements can be easily addressed by implementing a migration interface on the PCI PF through an administration queue and a command interface for migrating PCI VFs. Such an interface has broader applications beyond migration. Therefore, as part of the open standards community, several technical committee members have designed a virtio administration command and a queuing interface. Our design introduces two main building blocks.

1. The ability to get and set the device parts.
2. The ability to stop and resume the virtio device.

We also observe a similar demand in the industry for analogous design concepts, as indicated in [27], which supports the idea of migration based on the device state. Our findings indicate that our design achieves efficiency by utilizing Virtual Functions (VFs) or Scalable I/O Virtualization (SIOV) devices at each nested layer and their associated synthetic Physical Functions (PFs) through the design outlined in [26]. This approach ensures resource control and adheres to the principle of equivalency at all nesting layers by implementing hypervisor-level control for migration and pass-through devices to the guest VM. Although our design employs the PCI PF as the migration interface in a hypervisor, it is not confined to the PCI PF or the hypervisor itself. Future systems could delegate such tasks to other trusted PCI functions that possess migration capabilities and may not be directly accessible to the hypervisor. For instance, a PCI PF with a migration interface could be mapped as a pass-through Trusted Device Interface (TDI) to a Trusted Virtual Machine (TVM).

**Device parts**

A device state is composed of multiple device parts, which can be read or written by the migration interface. Our design facilitates the exchange of these device parts from the source to the destination during the migration phase. Each device part represents the runtime state of the device. Device parts are categorized in two ways:

1. Common device parts.
2. Device type-specific device parts.

Each device part is structured in a generic type-length-value (TLV) format, allowing each part to be of variable length and identified by its own type. A length field also enables the extension of the device part format in the future if necessary.

1. **Common device parts**
Virtio devices come in many different types. According to the virtio specification, there are at least 19 distinct device types. Common device parts define attributes that are shared across all device types. This approach prevents the duplication of definitions for common parts across all devices.

2. **Device type specific device parts**
This category of device parts is specific to a given device type. This enables extendibility of new feature for every device type.

Certain device parts are exchanged only for informational purposes and validation. A virtio VF device is provisioned with specific attributes that are read-only for the driver accessing the VF. These attributes are also read-only for the migration interface and are optional. The migration interface at the source hypervisor may choose to exchange these attributes, and the destination migration interface can verify them to ensure that they match exactly what is provisioned on the destination. This optional feature helps to detect errors in-band at an early stage of the device migration phase.

Certain stateful virtio devices such as virtio-fs may as well implemented the needed storage for maintaining the device state, hence a device parts based method for handling the device state on destination is no longer a problem for hardware based design as once thought in slide_41 in [13], for examples multiple DPUs [14], ]15] and [16] implements 64B, 16B and 16GB of DDR memory for accelerating the cloud networking workloads. Therefore, on demand consumption of device parts memory on the NIC during short device migration duration is optimized. Also, certain cloud platform [17] may support suspend and resuming the VM through the PCI VF power management which would also require storage of the device state in the device. In our design we abstracted this memory using the device get and set parts resources. The migration interface driver allocates the resources indicating to the device the type of memory to allocate and keep track of for migration.

## Device migration flow

Device migration flow involves several steps on both the source and destination hypervisor systems. Some steps occur while the VM is actively accessing the virtio-net device and performing high-bandwidth packet operations, while others take place after the device and the VM have been suspended.
For a single device migration, a typical flow is as follows:

1. The source hypervisor retrieves all device parts while the device and VM are running.
2. The destination hypervisor stops the device identified for migration and mapped to the guest VM.
3. The destination hypervisor configures all writable device parts on the destination device while it is stopped.
4. Steps 1 and 2 may be repeated multiple times by the hypervisor while the VM is still running, depending on the amount of system memory yet to be migrated. Steps 1 and 2 are often referred to as the pre-copy stage.
5. Once the source hypervisor decides to stop the VM for migration, it suspends both the VM and the device.
6. The source hypervisor then retrieves all the device parts one final time and transfers them to the destination hypervisor.
7. The destination hypervisor sets the device parts on the device and resumes its operation.

To summarize, a device migration downtime is a function latency of commands to:

1. Stop the source device.
2. Get all the device parts of the source device.
3. Set all the device parts on the destination device.
4. Resume the destination device.

In this design, the destination migration interface receives the device parts multiple times and is expected to set them multiple times. However, in most production cloud environments where applications are running user networking and compute workloads, there are no changes in the device parts between two consecutive readings. Consequently, except for the initial operation of setting the device parts, all subsequent settings effectively become no-ops, resulting in very minimal device setup time. This pre-copy technique, applied similarly to how a standard VM applies it to VM memory, achieved a 340% reduction in VM downtime compared to when pre-copy is disabled. More detailed performance analysis is discussed later in the paper.

In the rare event that the device parts are changed while being set for the second time, the device only applies changes with respect to the previous setting. This approach effectively amortizes the device setup latency.

## Software stack

Our migration interface design is using the VFIO [18] UAPI to achieve the passthrough device access to the guest VM by extending the existing vfio-virtio driver [19]. VFIO based passthrough has been a most common way in Linux kernel for more than a decade which supports implementing migration interface, which is implemented by many other PCI devices [20] and also offers a user space friendly interface in [21].
As shown in Figure 1, virtio net PCI VF is passthrough to the guest VM via VFIO fd. VFIO fd is composite software interface that consists of VF specific configuration space access, control + data plane access and migration interface. This UAPI implementation in the virtio-vfio driver interacts with the migration administration commands provided by the virtio PCI driver using administration queue.
This approach also uses the recently added iommufd [22] interface which collects the dirty page tracking information

from the CPU's iommu via the IOMMU driver, here the pages are the one which is accessed by the virtio net devices and also any other passthrough device to the same guest VM.

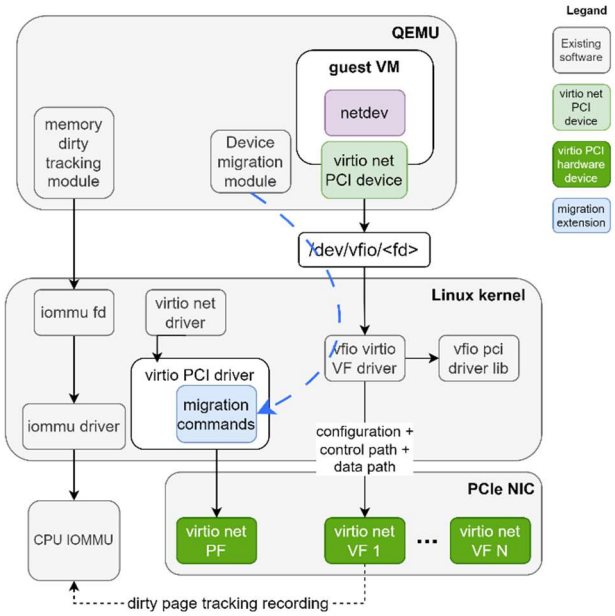


Figure 1. software stack using virtio VFIO driver for device migration and iommu fd for dirty page tracking.

## 4. Virtio PCI Device vs ConnectX vdpa

We analyzed a device migration for an Nvidia ConnectX-6DX vdpa, which is useful in many ways. However, it addressed limited use cases and requirements. We began our research by profiling the migration sequence, which involves the following operations:

1. Stop the source device.
2. Reconfigure the queues addresses again on the source device to use the new shadow virtqueue addresses located in the hypervisor memory.
3. Resume the source device.
4. Stop the source device.
5. Setup the queues and configuration on the destination device.
6. Resume the destination device.

Compared to the vdpa device-based migration, our device parts based designed has following differences:

(a) VM downtime reduction
Our design eliminated two latency sensitive operations on the source hypervisor. First, it eliminated the operation #2 and #3 and #4.

(b) When comparing the number of operations and their latency, it might be noted that queue reconfiguration and device setup are also performed in our design on the destination hypervisor. However, the main difference compared to the vdpa approach is:

A. In the vdpa approach, these traffic-disruptive reconfiguration operations are conducted on the source hypervisor while the VM is running. In contrast, our design performs these operations on the destination hypervisor, where the VM will be migrated without traffic disruption.

B. The hypervisor may intercept virtio driver notifications (also informally known as doorbells from the driver to the device) to capture and process the guest VM driver posted descriptors. To avoid race conditions, these notifications typically result in numerous VMEXIT calls during VM migration. Our design eliminates these VMEXITs.

(c) Eliminated vendor-specific driver:
Our design eliminated the Mellanox ConnectX-6DX specific vdpa driver on the hypervisor, which depended on a rapidly changing kernel and mlx5 core driver. This reduction significantly decreases the constant maintenance overhead. A cloud operator in the hypervisor now uses a single virtio driver for all virtio device types regardless of underlying producer of the virtio NIC.

(d) Hypervisor CPU utilization savings:
A guest VM equipped with a 100Gbps virtio net device at 1500 bytes packet rate, emits 8.3 million descriptors in each direction across 64 transmit and receive queues, all of which the hypervisor must track. This tracking accounts for an average of 4% of the CPU utilization dedicated to reposting the descriptors, which is a fair amount of CPU resource loss on the cloud system. These descriptors are already present in the guest VM virtio rings; hence our design didn't have to track any of it.

(e) Rather than using the hypervisor CPU for shadow virtqueue tracking, our design relies on the CPU/IOMMU memory write tracking (also known as dirty page tracking) capability to monitor memory written by the device and migrate it to the destination. This simplification extends to two different CPU platforms, Intel Sapphire Rapids and AMD Genoa. However, this benefit is not available on slightly older generation CPUs, which are widely deployed but lack the memory tracking capability.

(f) In the cloud deployment use case, we anticipate supporting NUMA topology-aware virtio VFs which are recently introduced as multi-PF NICs [4] for the guest VMs. Composing such PCI devices again from the QEMU layer is redundant, error-prone, and complex, especially when they are already available as part of the platform. Our proposed device migration design achieves such platform enhancements at no additional engineering cost because the device migration abstraction is implemented at the PCI device level.

## 5. Virtio PCI Device vs IDPF

We only compared the design of virtio migration interface vs the IDPF [25]. Both the designs are very similar in nature using VFIO subsystem.
They key difference from IDPF [25], is that our design utilized IOMMUFD based dirty page tracking mechanism and started with simple approach.
The second difference is, in our design we have bring the notation of device migration circuitry through an abstraction of resource objects. It is left to the NIC hardware vendors to implement such an object as memory, compute or mix of these resources in the hardware. Our design also exposes these resource capabilities of the migration interface to decide on when to fail migration on unavailability of the resources.
Since the device parts are a generic infrastructure, it can be used beyond migration interface for debugging as well as they are connected to the resource objects, where some resources can be used for migration and others for debugging. It is unclear from the IDPF [25], if the mailbox channel commands is generic enough for accessing the device parts.

## 6. Performance evaluation

We measured and analyzed the performance on various system level parameters. Our performance micro benchmarks were considering following parameters with active network traffic for bandwidth measurements with a link speed of 200Gbps.

1. 1 to N device scale per VM
2. 1 to M number of channels per netdevice with active traffic upto 200Gbps per VM.
3. 1 to K multiple vCPUs per VM
4. Varying amount of memory per VM to observe the effects on the dirty page tracking by the IOMMU and to observe any side effects of non-dirty pages.

System configuration:
CPU: AMD Genoa
QEMU: 9.1
Hypervisor OS: Linux kernel 6.8 + vfio virtio driver
Guest VM: Oracle Linux 8.4
Performance tool: iperf, iperf3

We analyze various performance test results with and without enabling the device pre-copy while keeping memory pre-copy enabled. This gave the insights into the benefits and contribution of device pre-copy in supporting multiple devices. Figure-3 shows when the number of devices is changed from 1 to 4 per guest VM, the VM downtime stayed in the boundary of 128msec to 334msec when device pre-copy is enabled. When the device pre-copy is enabled for 4 devices, each with 32 netdevice channels, the VM downtime decreased by 519% from 1735msec to 334msec compared to device pre-copy disabled.

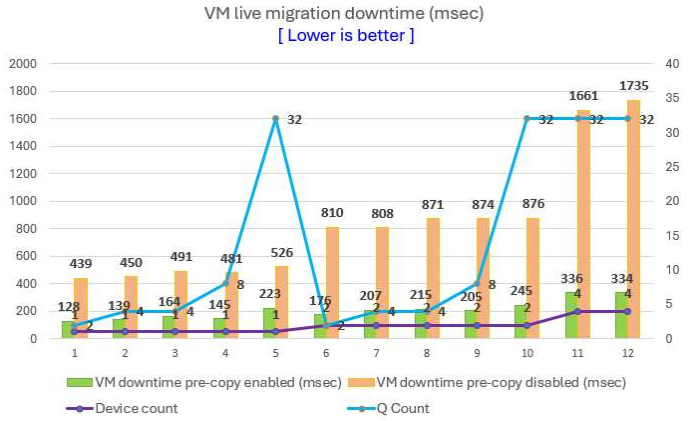


Figure 2. VM downtime with respect to number of devices and number of netdevice channels per VM, with device pre-copy enabled/disabled.

We also analyzed the effect on the total VM migration time with device pre-copy enabled and disabled with respect to other VM resources such as memory and vCPU and number of virtio net devices passthrough to the VM. Columns 10 and 11 in the Figure 3 shows that increasing the number of devices from 2 to 4 while keeping the same number of vCPUs and VM memory, total VM migration time nearly doubles when device pre-copy is enabled. This is because of setting up the device on the destination during the pre-copy phase. In the second observation, columns 11 and 12 in Figure 3 shows that increasing the VM memory from 64GB to 128GB while keeping the same device configuration and same workload also increases the total VM migration time. It can be concluded that either of increasing the VM memory or device configuration has equal effect on the VM total migration time, but no effect on the VM downtime due to pre-copy enablement. Interestingly with the pre-copy disabled, for smaller CPU count and memory, there is nearly no impact on the total migration time of a VM, however as the CPU, memory and device count increases beyond 16 vcpus and 16GB memory, with larger queue count, the VM migration time increases with larger resource usage.

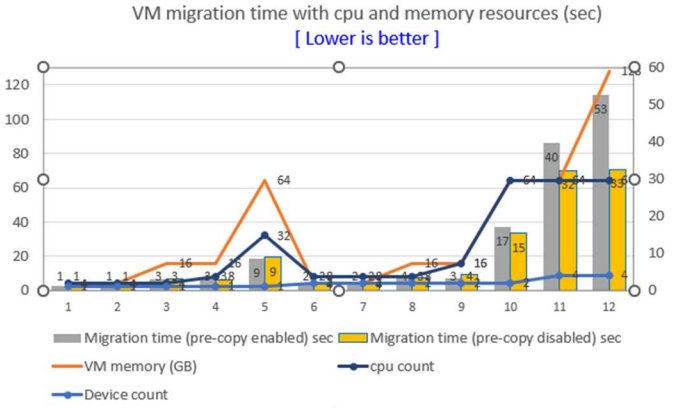


Figure 3. VM downtime with respect to device count, vCPUs, VM memory, and netdevice channels per VM, with device pre-copy enabled/disabled.

We further analyze the latency of key system-level operations that contribute to VM downtime using a VFIO virtio driver extension and QEMU profiling. Table 1 shows the latency with device-level pre-copy disabled mode, while Table 2 shows the latency with device-level pre-copy enabled mode. Both tables list the latency of each operation that contributes more than 1 millisecond of latency, ignoring smaller latency contributors.

Our observations indicate that enabling device-level pre-copy results in more pre-copy operations of the system RAM and device state. This leads to a 50% reduction in memory (RAM) copy time in the device pre-copy method in the micro-benchmark results, saving 100 milliseconds of downtime.

Additionally, with pre-copy enabled, the driver receives hints through pre-copy information via ioctl(). This allows the driver to read the device state and cache it as soon as the device state reaches the PRE_COPY state. However, reading the device state asynchronously would further help to reduces the 8msec latency to near zero. This is because such latency bound commands can easily overlap with RAM migration phase utilizing the asynchronous queuing interface. Consequently, the final state reading time is reduced to a negligible 0.012 microseconds due to the caching in the driver, representing an improvement of 16,666 times on the source hypervisor compared to a pre-copy method. However, this latency reduction is not proportionately reflected in the VM downtime due to high latency exists in device memory copy.

| Location | Operation | Latency (msec) |
|---|---|---|
| Source hypervisor | Device state change from RUNNING to RUNNING_P2P | 7 |
| | Device state change to STOP | 0.6 |
| | Save (read) the system RAM | 230 |
| | Save (read) the device state (per device) | 20 |
| Destination hypervisor | Load (write) the system RAM | 226 |
| | Load (write) the device state (per device) | 300 |
| | Device state change to STOP to RUNNING_P2P | 1.1 |
| | Device state change from RUNNING_P2P to RUNNING | 2 |

Table 1. VM downtime breakdown when the device pre-copy disabled.

| Location | Operation | Latency (msec) |
|---|---|---|
| Source hypervisor | Device state change from RUNNING to PRE_COPY_P2P | |
| | Device state change to STOP_COPY | 8 |
| | Save (read) the system RAM | 135 |
| | Save (read) the device state (per device) | 0.012 |
| Destination hypervisor | Load (write) the system RAM | 131 |
| | Load (write) the device state (per device) | 11 |
| | Device state change to STOP to RUNNING_P2P | 1.1 |
| | Device state change from RUNNING_P2P to RUNNING | 2 |

Table 2. VM downtime breakdown when the device pre-copy enabled.

We observed that device handling commands are performed serially by the user space migration module in synchronous system call read() and write() even though the device state of multiple devices are unrelated to each other. Hardware devices are inherently parallel in nature to support multiple outstanding commands per administration queue. The synchronous user space was unable to utilize the hardware parallelism for such use case. Though we believe that io_submit() system call usage using aio_read() and aio_write() system call extension for the VFIO subsystem can further improve by queuing these operations to the device. Such extension would not be possible using ioctl() UAPI which are designed mainly for synchronous operations from the user space.

## 7. Lessons learnt and next steps

1. Device parts exchanged during pre-copy and stop copy phase of the VFIO state machine are nearly same. This effectively performs de-duplication of the state on the destination device when setting the device state. This simplified the device migration design. We also considered an alternative where such a deduplication is done on the source device. Such method can be useful if the device state becomes very large and often changing. In all known cloud operators known workload of near term and with upcoming virtio net features enhancements, the de-duplication approach on the destination device seems to fit the downtime requirements even at moderate scale of 4 to 8 devices per VM.

2. A generic administration command and queue interface was originated from the migration interface need; however, it became useful beyond it. We were able to accelerate even virtio legacy 0.9.5 workload VMs too and it also overcome the limitation of the PCIe SR-IOV which never supported IO-BAR but were part of the virtio specification. Community finds administration interface even further useful to implement flow filters functionality [6].

3. CPU IOMMU dirty page tracking is only available on the latest CPUs even though the CPU specifications have it listed it for some time. Cloud operator's workload runs on few CPU generations, where all CPU family may not support dirty page tracking. Hence, virtio specification incorporating and optionally supporting device side dirty page tracking functionality can be useful though not a must.
4. We began to develop specification and implementation in tandem that resulted in measuring trade-offs between various approaches more effectively, such as deduplication of device parts on destination device vs source device.
5. The latest and revised draft specification for this design is posted at [21] for the community and technical committee review. Our design has been developed and validated for various sizes of the VMs and multiple devices per VM as listed in the performance tests. After community acceptance of the virtio specification, we would like to open source the device migration driver extension to be included in the upstream Linux kernel under GPL license as part of the existing virtio VFIO driver [19].

## 8. Conclusion

1. Our design is first of its kind for the virtio PCI hardware devices. It is easily extendible to multiple types of virtio devices with just new definitions of the device parts like Linux netlink messages [20].
2. A device pre-copy approach achieves 340% reduction in the VM downtime for a single device, and 494% reduction in downtime for multiple devices with large number of netdevice channels. This high performance is achieved at nearly no additional memory burden from the device.
3. IOMMU based dirty page tracking enablement has no effect on the VM migration time or VM downtime. Instead, it simplified the device migration design without using PCI PRI, PASID, and without any vendor or device specific software-based queue tracking approach.
4. Our design does not require any support of complex PCI PRI interface for dirty page tracking as imagined in slide_46 of [13].
5. Modern purpose built DPUs [14], [15] and [16] used in accelerating cloud workload are equipped with memory of 64GB, 16GB and 16GB respectively, which can dynamically store the required device parts either on the source or destination device during the pre-copy phase of device migration. Virtio specification abstracting it using a resource object is a good start, which may require more granular approach in future the specification and eco-system matures.

## Acknowledgements

Authors like to thank several reviewers who provided invaluable feedback and insightful comments of our design and draft specification that resulted in more robust specification, namely Michael S. Tsirkin @RedHat, Jason Wang @RedHat, Jason Gunthorpe @Nvidia, Max Gurtovoy @Nvidia, Ben Walker @Nvidia, Chaitaniya Kulkarni @Nvidia and Si-Wei Liu @Oracle.

## References

1. virtio specification, https://docs.oasis-open.org/virtio/virtio/v1.3/csd01/virtio-v1.3-csd01.html#x1-1070001
2. Vdpa Linux kernel overview, https://www.redhat.com/en/blog/introduction-vdpa-kernel-framework
3. Nvidia virtio PCI devices, https://docs.nvidia.com/networking/display/bluefieldvirtioetv190/introduction
4. VFIO device migration, https://www.qemu.org/docs/master/devel/migration/vfio.html
5. Multi-PF NICs, https://netdevconf.info/0x18/sessions/talk/multi-pf-single-netdev.html
6. Flow filters for virtio net device, https://lore.kernel.org/virtio-comment/20240604132903.2093195-1-parav@nvidia.com/T/#t
7. Open compute SIOV specification 1.0 http://files.opencompute.org/oc/public.php?service=files&t=4bc59a2c8e894b2093d792b8766925d7
8. RDMA in virtual environments https://www.openfabrics.org/images/docs/2013_Dev_Workshop/Wed_0424/2013_Workshop_Wed_0830_BhaveshDavda-vRDMA.pdf
9. NVMe Virtualization for Cloud Virtual Machines, ICPE '22, April 9–13, 2022, Bejing, China, https://dl.acm.org/doi/pdf/10.1145/3489525.3511688
10. PCI peer to peer DMA support, https://docs.kernel.org/driver-api/pci/p2pdma.html
11. Intel IOMMU specification, https://cdrdv2-public.intel.com/671081/vt-directed-io-spec.pdf
12. AMD IOMMU specification, https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/specifications/48882_IOMMU.pdf
13. vdpa-net Live Migration with Shadow virtQueue, https://kvm-forum.qemu.org/2023/vDPA_sw_lm_-_KVM2023_6Ix6R5i.pdf
14. AMD Pensando Ela Data Processing unit product brief, https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/product-briefs/pensando-elba-product-brief.pdf
15. Nvidia Bluefield-3 DPU, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf
16. Intel IPU Platform F2000X-PL product brief, https://cdrdv2.intel.com/v1/dl/getContent/792306
17. Google cloud with suspend and resume functionality, https://cloud.google.com/compute/docs/instances/suspend-resume-instance
18. VFIO Linux kernel UAPI, https://elixir.bootlin.com/linux/v6.9.3/source/include/uapi/linux/vfio.h
19. virtio vfio device driver, https://elixir.bootlin.com/linux/v6.9.3/source/drivers/vfio/pci/virtio
20. Multiple PCI devices implementing VFIO interface, https://elixir.bootlin.com/linux/v6.10-rc3/source/drivers/vfio/pci
21. John Johnson, Jagannathan Raman, Remote device Emulation using VFIO, https://static.sched.com/hosted_files/kvmforum2021/6a/Johnson_Raman_Ufimtseva_Vfio-user.pdf
22. 2. Eric Auger, Yi Liu, Jason Gunthorpe, Kevin Tian, IOMMUFD Discussion, https://lpc.events/event/16/contributions/1325/attachments/1014/1953/LPC2022_iommufd.pdf
23. virtio device parts handling commands proposal, https://lore.kernel.org/virtio-comment/20240601145042.2074739-1-parav@nvidia.com/T/#t
24. Linux kernel netlink message format, https://docs.kernel.org/userspace-api/netlink/intro.html#nl-msg-type
25. IDPF live migration support, https://netdevconf.info/0x17/docs/netdev-0x17-paper30-talk-slides/idpf_live_migration_support.pdf
26. Yui Washizu, "Unleashing SR-IOV Offload on Virtual Machine", https://netdevconf.info/0x17/docs/netdev-0x17-paper20-talk-slides/Netdev_0x17.pdf
27. Si-Wei Liu, "Hardware Friendly Vhost vDPA Towards an Efficient and Migratable Device Model", https://static.sched.com/hosted_files/kvmforum2022/3a/KVM22-Migratable-Vhost-vDPA.pdf
28. Formal Requirements for Virtualizable Third Generation Architectures, https://dl.acm.org/doi/pdf/10.1145/361011.361073

## Authors Biographies

Satananda Burla is a Senior Principal Engineer at Marvell, involved in design and development of Marvell Octeon SoC architecture. He also leads the design and implementation of Networking, Security and Storage solutions with Marvell Octeon SoCs. He represents Marvell at OPI, OASIS Virtio, UEC and other technical forums.

Parav Pandit, Linux kernel contributor in area of RDMA, net device stack, device drivers, cgroup and a software architect for virtualization of NIC and DPUs working at Nvidia.

Yishai Hadas, Linux kernel contributor in various areas of RDMA, VFIO, live migration, device drivers working at Nvidia.

Feng Liu, working for Nvidia, is mainly responsible for the virtio software development of BlueFeild DPU product. With a rich background that includes roles at Baidu and ZTE, Feng has experience in Linux kernel development, Linux driver development and other DPU products.

Avihai Horon, is open-source contributor in various areas of live migration, QEMU, VFIO, device drivers working at Nvidia.