Journey of advancing virtio live migration

Parav Pandit, Satananda Burla, Avihai Horon, Feng Liu, Yishai Hadas

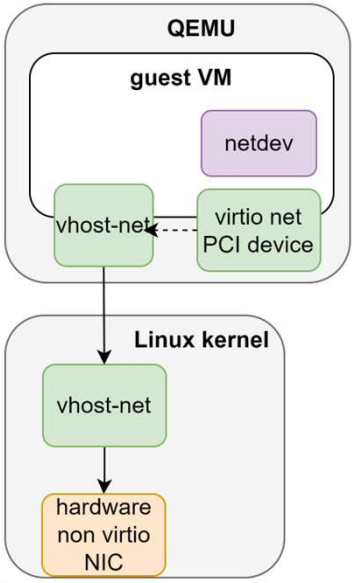
Agenda

- Overview and Goals
- Design principles
- Design
- Comparing solutions
- Performance results
- Summary

Virtio offerings to guest VM

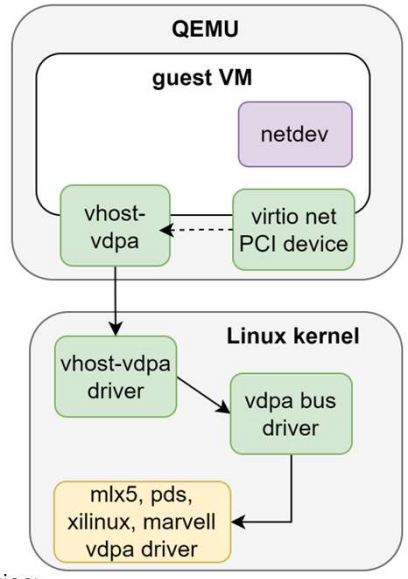
Overview

Method-1 (software)



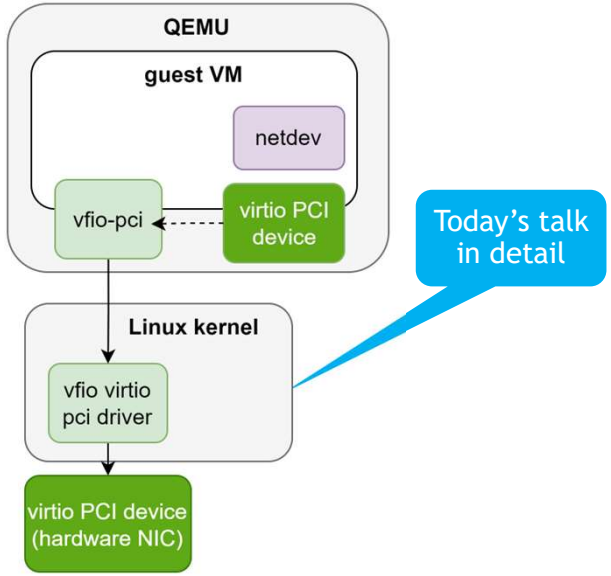
1. Device:
   1. Composed by QEMU virtio PCI module
2. UAPI: vhost-net
3. Perf: CPU bound
4. Migration: user space virtio

Method-2 (vdpa kernel)



1. Device:
   1. Composed by **virtio** PCI + vhost-vdpa
2. UAPI: vhost-vdpa kernel
3. Perf: hardware bound + hypervisor cpu bound
4. Migration: user space virtio + vendor driver + qemu + kernel modules

Method-3 (vfio kernel)



1. Device:
   1. Compose generic only PCI config space + msix in QEMU
2. UAPI: vfio-pci, common for all PCI devices
3. Perf: hardware bound
4. Migration: vfio kernel, common for all virtio devices

MARVELL NVIDIA

Goals

1. Achieve device migration for virtio PCI VF hardware devices

2. Achieve lowest possible VM migration downtime (< 200msec)
   - At scale of 1 to 4 devices with hundreds of tx+rx queues
   - At 64 to 128 vCPUs
   - At 128 GB VM memory

3. Single data path and migration framework for virtio net/blk/fs/console devices

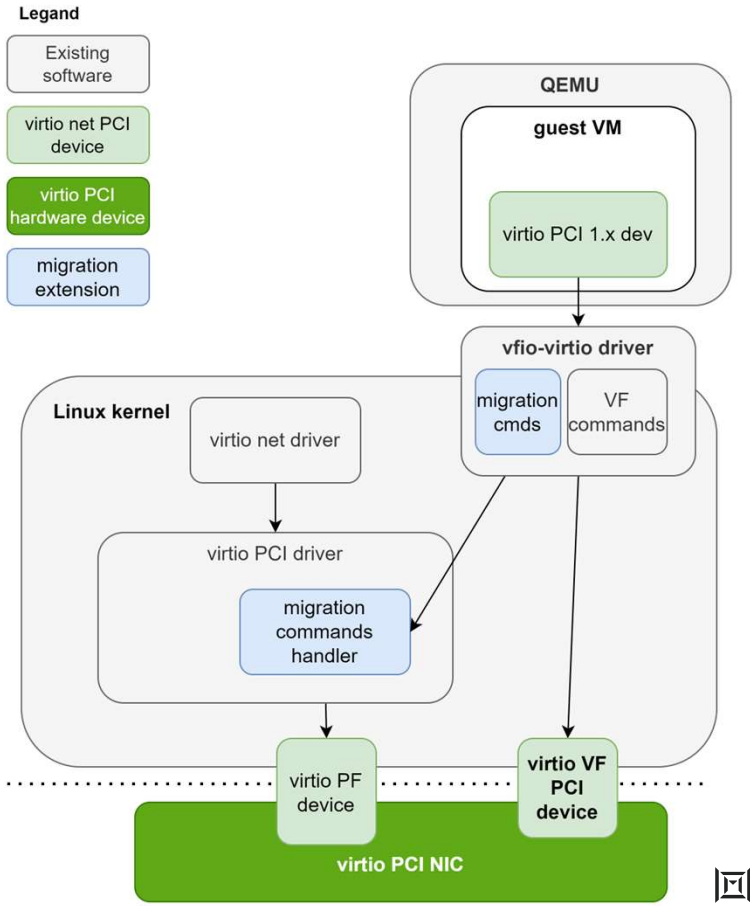
4. Virtio feature extensions in the device.

Virtio live migration design
Design principles

1. Zero-lines of code change on hypervisor driver to add a new virtio feature
2. Near O(1)* downtime regardless of number of devices in the guest VM
3. Least number of operations on src and dst hypervisors for live migration
4. Support seamless virtio device and PCI FLR reset from guest VM during live migration
5. Migration of SR-IOV VF and future SIOV_R2 in unified manner
6. Unified migration driver (and device) design regardless of virtio device type (net, blk, fs)
7. Unified design for virtio, nvme, mlx5 and similar industry devices
8. Utilize dirty page tracking either from (a) CPU/platform IOMMU or (b) from the virtio device
9. Support PCI P2P
10. Pave road for upcoming CC (Confidential Compute), IDE and TDISP

Virtio live migration design
Design (communication)

- VM to VF access:
  - SR-IOV VF passthrough to the guest VM
  - One or more VFs passthrough
- Migration driver:
  - Parent PF driver is running migration commands
  - PF runs migration communication channel
- Migration interface:
  - Supports multiple VFs migration at same time
  - Asynchronous interface via AQ
  - Energy efficient
  - Scalable by commands & by queue count
  - Enables Large device state transfer
  - Enabler for legacy acceleration



Linux software stack view

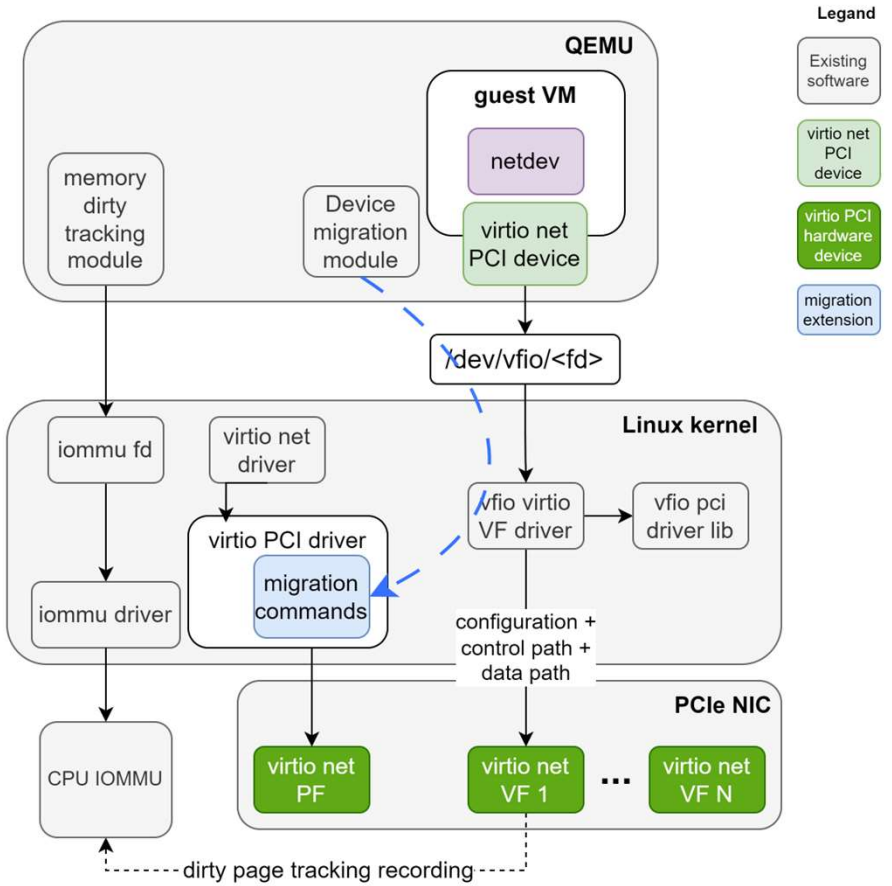
System level view

common UAPI across all device types

- ~~Virtio~~ common UAPI:
  - VFIO PCI driver passthrough + migration
  - QEMU only intercepts PCI config space and MSIX-Table
  - VFIO virtio driver and VFIO PCI driver/lib share common code

- CPU IOMMU:
  - Tracks the pages written by the PCI device
  - IOMMU driver tracks the page table entries
  - Iommu FD driver provides the UAPI

Device type agnostic UAPI



Virtio live migration design
Design (operations and parts)

- Device mode handling
  - Stop
  - Resume
- Device parts
  - Represents the device state
  - Agnostic to hardware supplier
  - Cloud operator choosen
  - Generation agnostic (NICv1, NICv2)
  - Common parts for net, blk fs
  - Device type specific parts
  - Wrapped in a resource objects to provide hints to the device
- Device parts commands
  - Get parts
  - Set parts
  - Get parts metadata
- Pre-copy support
  - Device parts exchange while src hypervisor running

- Typical pre-copy sequence:
  1. Destination HV:
     1. Stop the device
  2. Source HV:
     1. Get device parts one or more times
     2. Exchange device parts with destination HV
  3. Destination HV:
     1. Set device parts one or more times
  4. Source HV:
     1. Suspend the device
     2. Get final device parts
  5. Destination HV:
     1. Set final device parts
     2. Resume the device

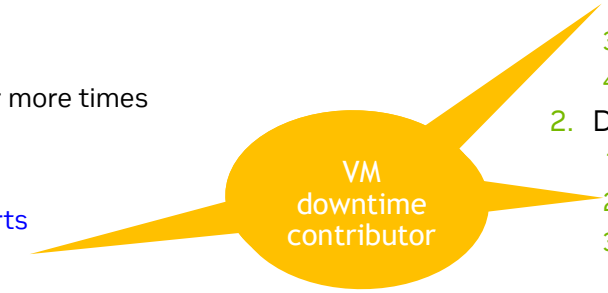
VM running (pre-copy)

VM downtime

MARVELL NVIDIA

Virtio live migration design
Migration sequence

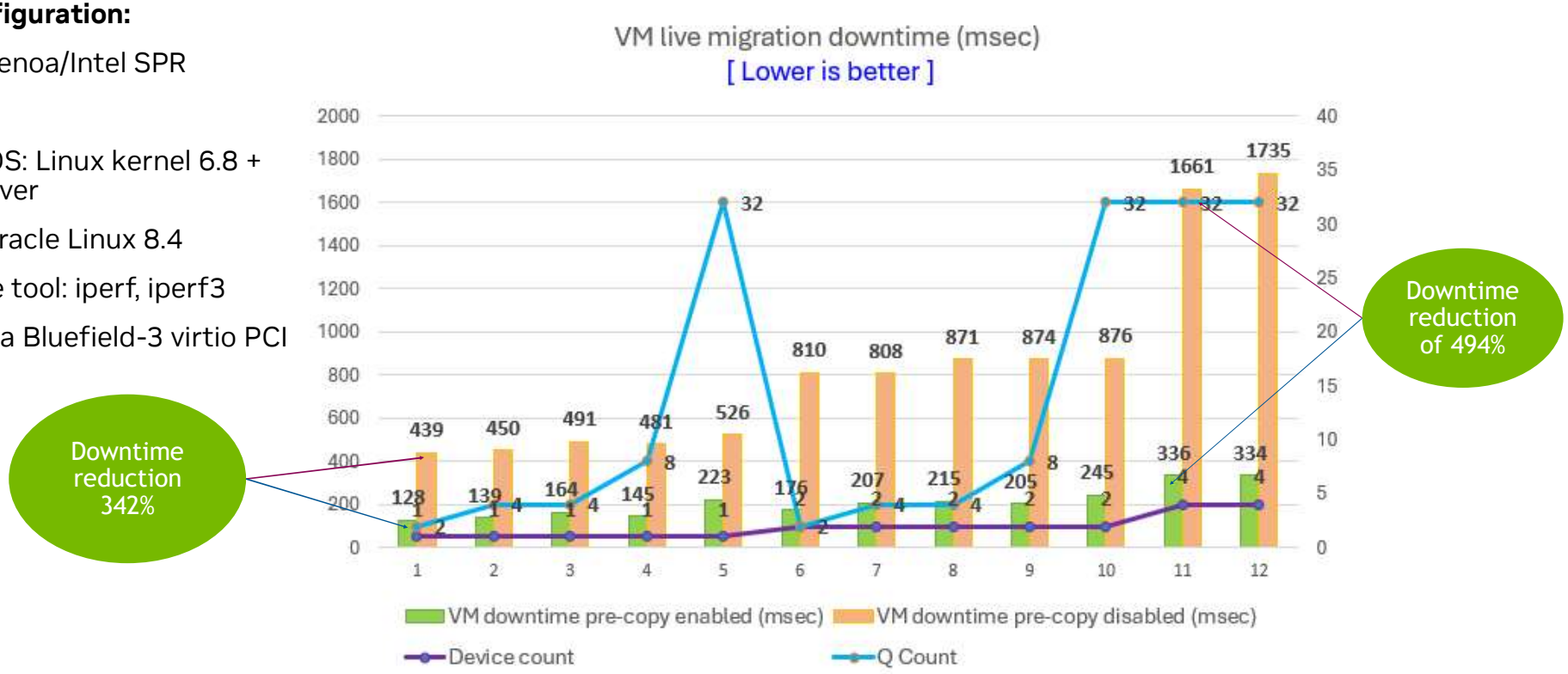
- Typical pre-copy sequence:
  1. Source HV:
     1. Get device parts one or more times
     2. Exchange device parts with destination HV
  2. Destination HV:
     1. Stop the device
     2. Set device parts one or more times
  3. Source HV:
     1. Stop the device
     2. Get the final device parts
  4. Destination HV:
     1. Set the final device parts
     2. Resume the device



- Typical kernel vdpa sequence:
  1. Source HV:
     1. Stop/suspend the device
     2. Reconfigure the queue addresses for shadow virtqueue
     3. Resume the device (and queues)
     4. Stop/suspend the device
  2. Destination HV:
     1. Setup queues
     2. Setup the configuration
     3. Start the device

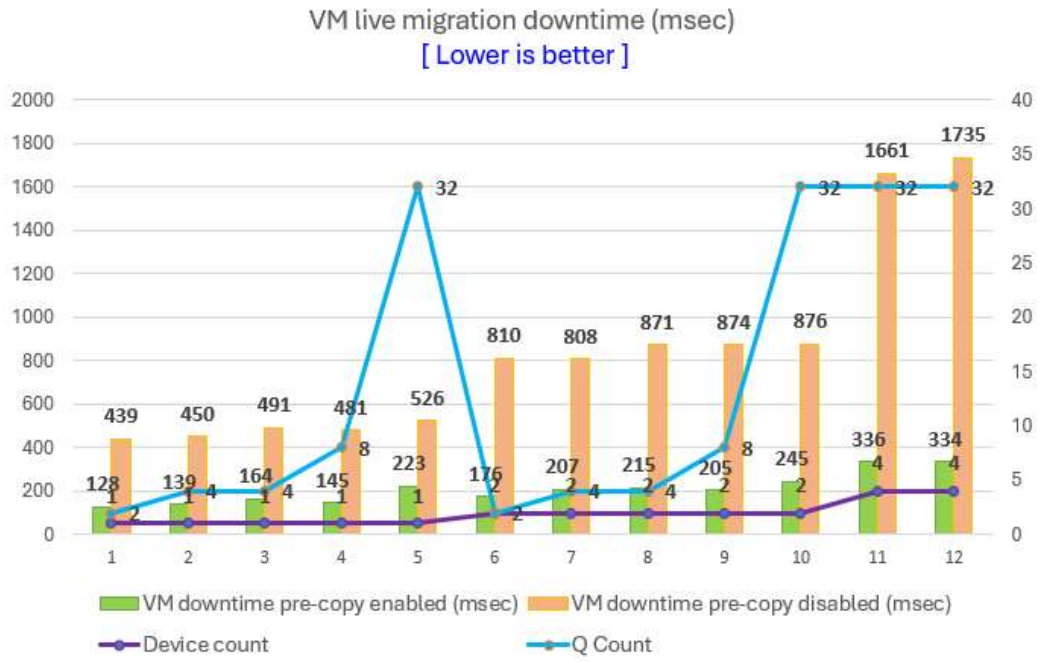
Performance analysis
Device scale vs VM downtime

- **System configuration:**
- CPU: AMD Genoa/Intel SPR
- QEMU: 9.1
- Hypervisor OS: Linux kernel 6.8 + vfio virtio driver
- Guest VM: Oracle Linux 8.4
- Performance tool: iperf, iperf3
- Device: Nvidia Bluefield-3 virtio PCI NICs



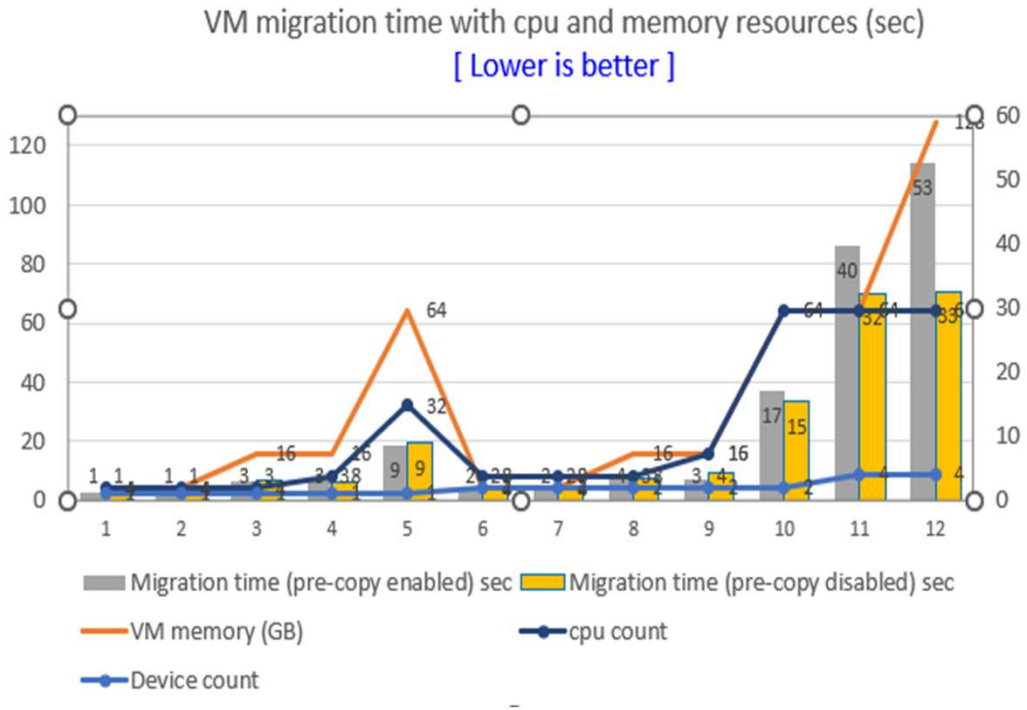
Performance analysis
summary

- Without device pre-copy method:
  - VM downtime is function of IO commands latency.
  - VM downtime depends on device resources.
- With device pre-copy method:
  - single device per VM, achieves 342% downtime reduction
  - 4 devices per VM each with 32 queues, achieves 494% downtime reduction.
  - VM downtime stays in range of 128msec to 334msec with resource scaling from 1 queue to 128 queues.
- IOMMU based dirty page tracking:
  - Does not reduce the performance with memory scale
  - Near zero effect on VM memory size (64GB to 128GB)



MARVELL NVIDIA

Performance analysis
Migration time vs pre-copy methods



- In pre-copy mode:
  - VM migration time is proportional to number of devices, VM memory.
- Without pre-copy mode:
  - VM migration time is relatively lower due to less iterations of device state management.

Performance analysis

Latency tasks during VM downtime

Device pre-copy disabled

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

Device pre-copy enabled

Location	Operation	Latency (msec)
Source hypervisor	Device state change from RUNNING to PRE_COPY_P2P	7
	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

Summary

- Design:
  - A device type agnostic VFIO UAPI
  - Simple yet efficient design that enables pre-copy giving 3x or higher VM downtime reduction
  - Implementation has more scope of optimization:
    - overlapping compute vs IO operations,
    - asynchronous and parallelizing it.
  - Fully transparent and managed by the hypervisor
  - Device state definition is agnostic of virtio device provider

- Specification status:
  - Approved by the OASIS technical committee + community
  - More device specific TLVs to be added.
  - Fixes in progress.

- Linux kernel upstream driver:
  - Formalize the vfio-virtio driver for kernel inclusion.

Acknowledgements

- Michael Tsirkin@Redhat for simplifying, extendible design
- Jason Wang @Redhat for solidify the synchronization aspects
- Si-Wei @Oracle for the presenting the core device state idea at KVM forum
- Jason Gunthrope, Max Gurtovoy, Gal Shalom, Chaitainya Kulkarni, Ben Walker @Nvidia in shaping the design

MARVELL NVIDIA