Improve VNF safety with Vhost-User/DPDK IOMMU support

No UIO anymore!

Maxime Coquelin
Software Engineer

KVM Forum 2017

AGENDA

- Background
- Vhost-user device IOTLB implementation
- Benchmarks
- Future improvements
- Conclusion - Questions

Background

Why do we need IOMMU support?

Background

Current status (i.e. without IOMMU support):

- Use of UIO or VFIO with enable_unsafe_noiommu_mode = 1

  → Taints the Kernel, require rebuild with some distros

  → Use of GPA (guest physical addresses) for virtqueues and buffers

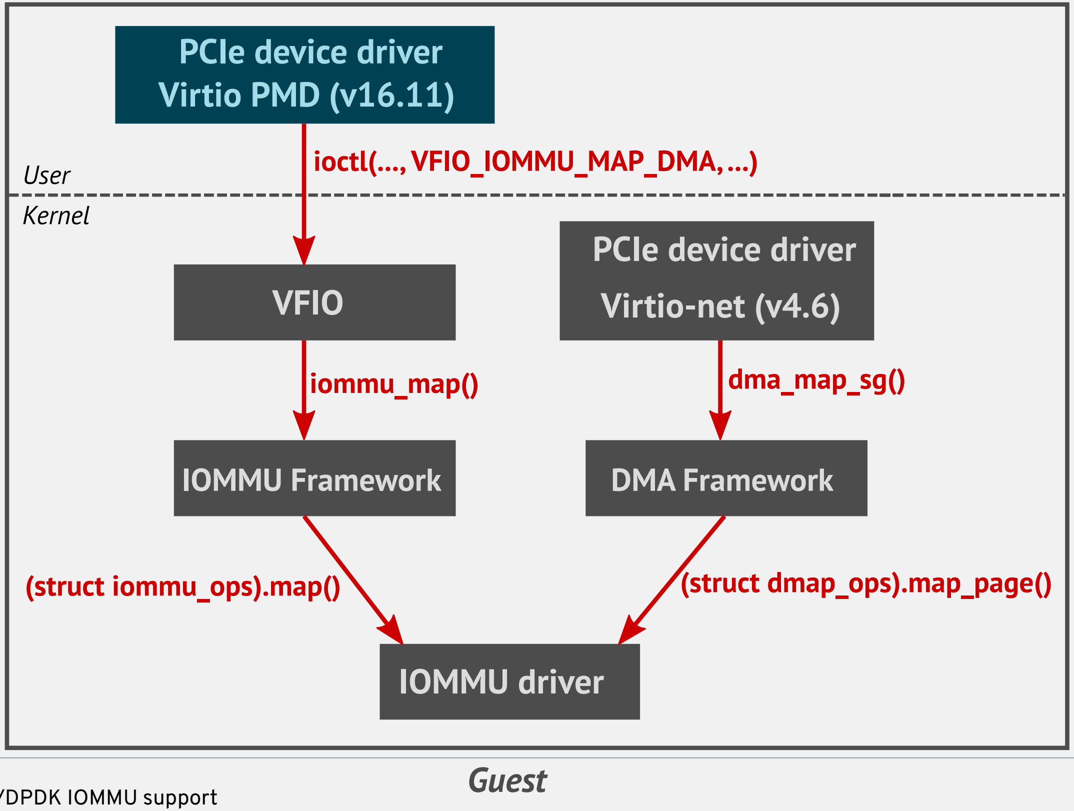
- Vhost-user backends mmaps all the guest memory with RW permissions
- DPDK app in guest could make Vhost to access memory the app hasn't access to

  → The guest app could pass random GPA as descriptor buffer address

  → Vhost backend overwrites random memory with packet content, or leaks random memory as a packet.

IOMMU support in guest

Background



Static vs. dynamic mappings

Background

We consider two types of DMA mappings

- Dynamic mappings (Kernel/Virtio-net driver)
  - → At least one dma_map()/dma_unmap() per packet
  - → At least one IOTLB miss/invalidate per packet
- Static mappings (DPDK/Virtio PMD)
  - → Single iommu_map/unmap() for all the memory pool at device probe/remove
  - → Only IOTLB misses the first time pages are accessed

vIOMMU support in Qemu

Background

- Emulated IOMMU devices implementations in QEMU
  - → x86 and PowerPC supported, ARM on-going
  - → Platform-agnostic Virtio-IOMMU device spec being discussed
- Provides IO translation & device isolation as physical IOMMUs do
- Generic IOTLB/IOMMU API provided in QEMU
  - → get IOTLB entry from (IOVA, perm)
  - → IOMMU notifiers (MAP/UNMAP)

vIOMMU support for Vhost backend dev in QEMU

Background

- Initially introduced with kernel backend
- Implements Address Translation Services (ATS) from PCIe spec

  → Using QEMU's IOTLB/IOMMU APIs
- Vhost-backend changes
  - Notify the backend for IOTLB invalidates
  - Notify the backend for IOTLB updates
  - Handle backend IOTLB miss requests

vIOMMU support for Vhost backend in kernel

Background

- Implements new protocol using Vhost-kernel chardev reads/writes
  - Other vhost-kernel requests uses ioctls
  - Required to be able to poll for IOTLB miss requests
- struct vhost_iotlb_msg message types
  - VHOST_IOTLB_MISS : Request QEMU for an IOTLB entry
  - VHOST_IOTLB_UPDATE : Update Kernel with a new IOTLB entry
  - VHOST_IOTLB_INVALIDATE : Notify Kernel an IOTLB entry is now invalid
- Device IOTLB implemented in vhost kernel driver
  - Relies on interval tree for better cache lookup performance → O(log(n))
  - Dedicated cache for virtqueues metadata → O(1)

Vhost-user device IOTLB implentation

Vhost-user protocol update

Vhost-user device IOTLB implementation

- Goal : design as close as possible to vhost-kernel protocol
- Problem : IOTLB miss request requires slave initiated requests support
  - But vhost-user socket only supports master initiated requests

  → Introduction of a new socket for slave requests
- Slave request channel
  - VHOST_USER_PROTOCOL_F_SLAVE_REQ protocol feature
  - VHOST_USER_SET_SLAVE_REQ_FD request to share new socket's fd
  - Re-use master's message structure, with new requests IDs
  - Only used by IOMMU feature for now

redhat.

Vhost-user protocol update (cont'd)

Vhost-user device IOTLB implementation

- IOTLB protocol update (Since QEMU v2.10, Vhost-user spec for details)
  - Master initiated : VHOST_USER_IOTLB_MSG

    → IOTLB update & invalidation requests

    → Same payload as vhost-kernel counterpart (struct vhost_iotlb_msg)

    → Reply-ack mandatory
  - Slave initiated : VHOST_USER_SLAVE_IOTLB_MSG

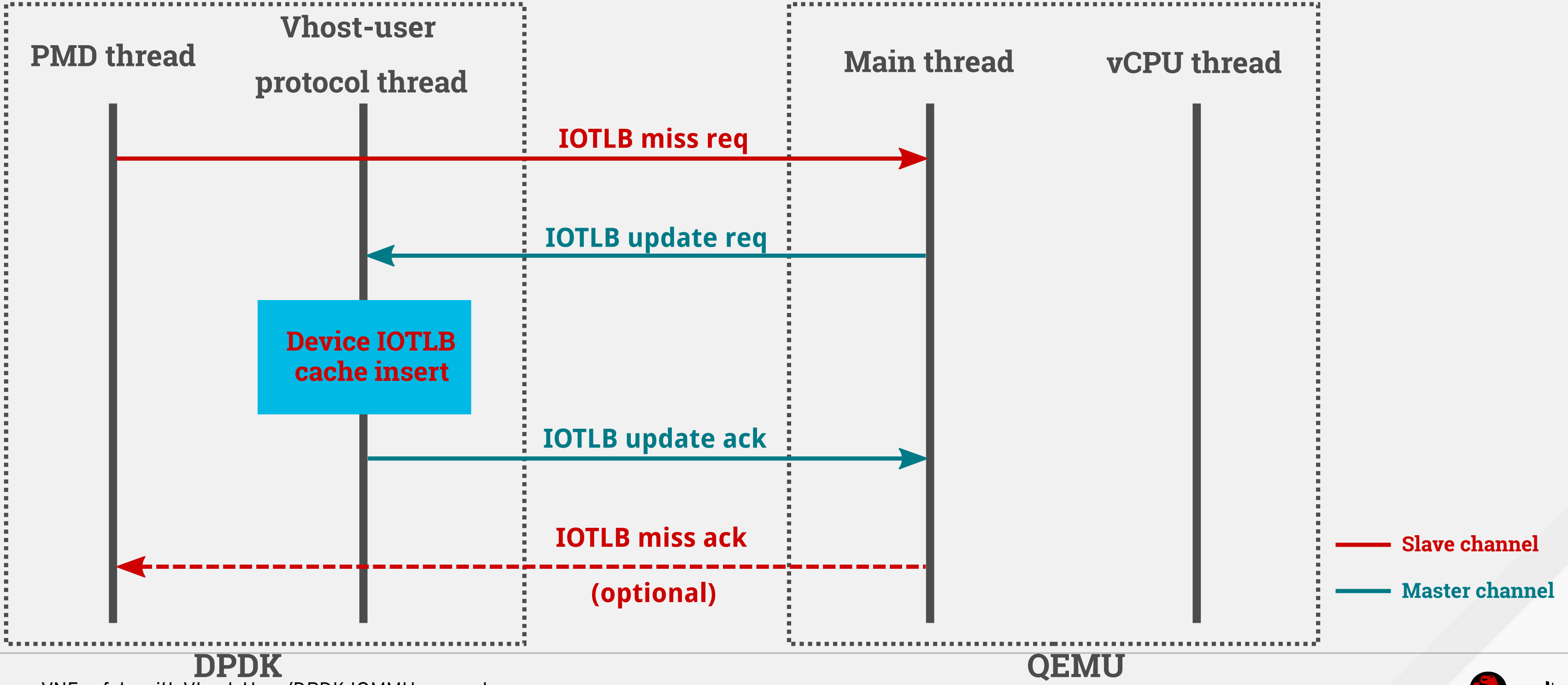
    → IOTLB miss requests

    → Also using struct vhost_iotlb_msg as payload

    → Reply-ack optionnal

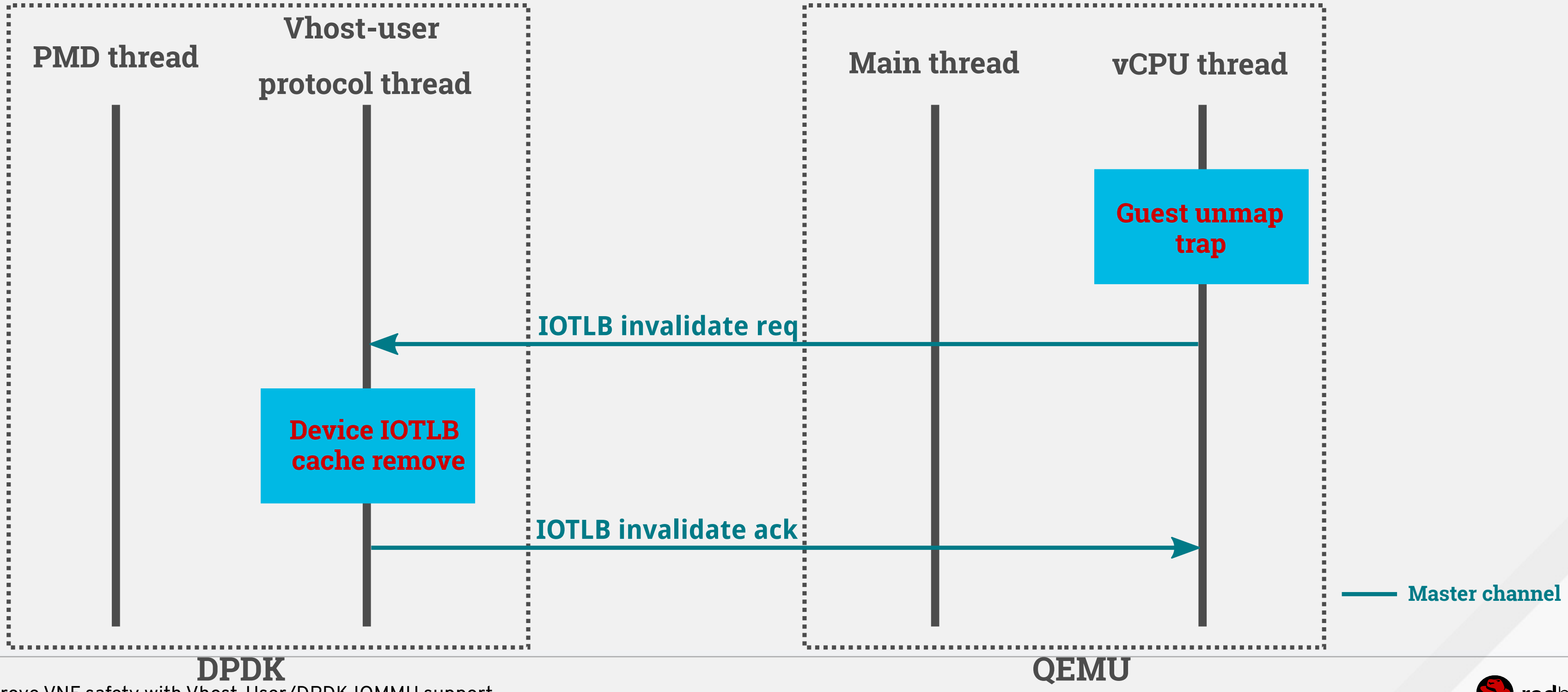
IOTLB miss/update sequence

Vhost-user device IOTLB implementation



IOTLB invalidate sequence

Vhost-user device IOTLB implementation



IOTLB cache implementation

Vhost-user device IOTLB implementation

- Device IOTLB cache implemented in Vhost-user backend

  → Avoid querying for every address translation
- Single writer, multiple readers to the IOTLB cache
  - Writer : Vhost-user protocol threads (IOTLB updates/invalidates)
  - Readers : PMD threads (IOTLB cache lookups)
  - Great case for RCU! Prototyped and tested, but…

    → liburcu is LGPLv2, only small functions can be in-lined

    → Adds dependency to DPDK build

    → Some distros don’t ship liburcu

IOTLB cache implementation

Vhost-user device IOTLB implementation

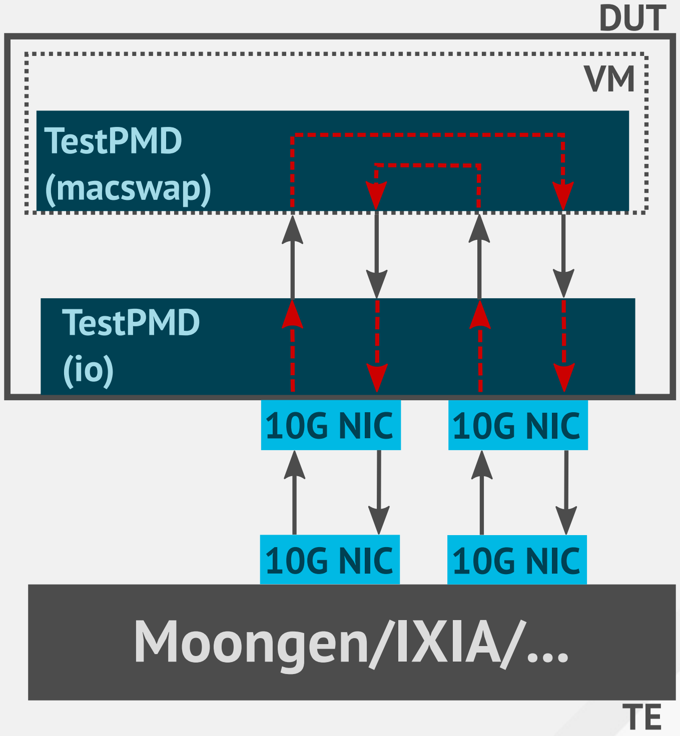
- Fallback : readers-writers locks (rte_rwlock)
  - Better than regular mutexes
  - But read lock uses rte_atomic32_cmpset(), optimizations to reduce its cost :
    - → Per-virtqueue IOTLB cache
    - → Read lock taken once per packets burst
- Initial cache implementation based on sorted lists
  - Not efficient, but enough with 1G pages.
  - Need a better implementation for smaller pages
- Cache sized large enough not to face misses with static mappings
  - → IOTLB cache evictions should only happen with buggy/malicious guests

Benchmarks

Physical → Virtual → Physical

Benchmarks

- PVP benchmark based on TestPMD
  - IO forwarding on host side
  - MAC swapping in guest to access packet header
- Setup information
  - T-Rex + binary-search.py from lua-traffigen
  - DUT
    - E5-2667 v4 @3.20GHz (Broadwell)
    - 32GB RAM @2400MHz
    - 2 x 10G Intel X710



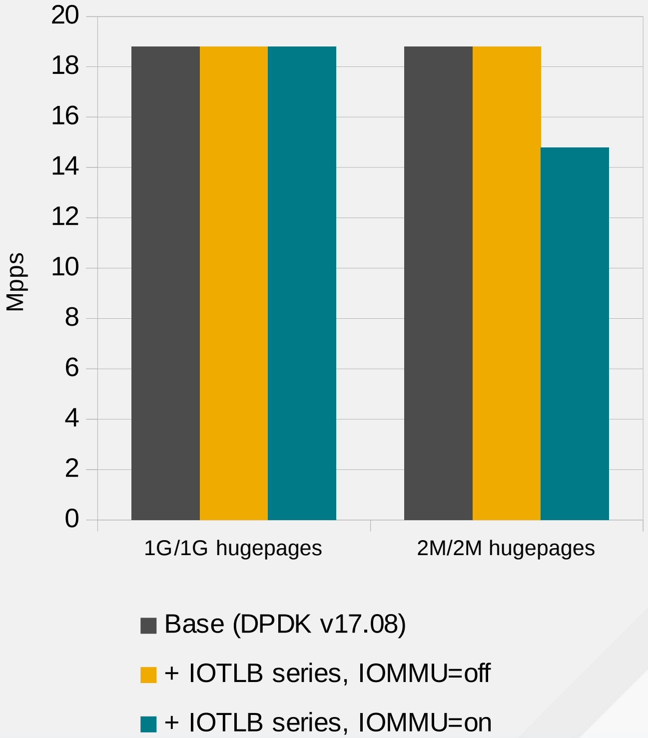
Physical → Virtual → Physical

Benchmarks

- PVP reference benchmark with IOTLB series v2
- Parameters: 64B packets, 0.005% acceptable loss, bidirectionnal testing (result is the sum)
- 2M/2M hugepages
  - IOMMU off : No performance regression
  - IOMMU on : ~25% degradation

    → IOTLB cache lookup overhead
- 1G/1G hugepages
  - IOMMU on/off : No performance regression

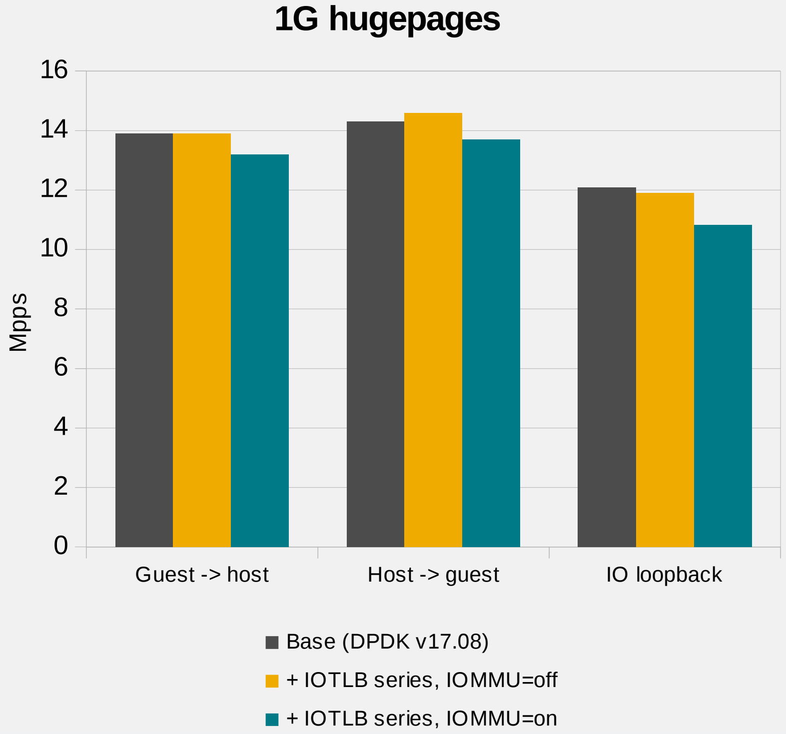
    → Virtio PMD is the bottleneck

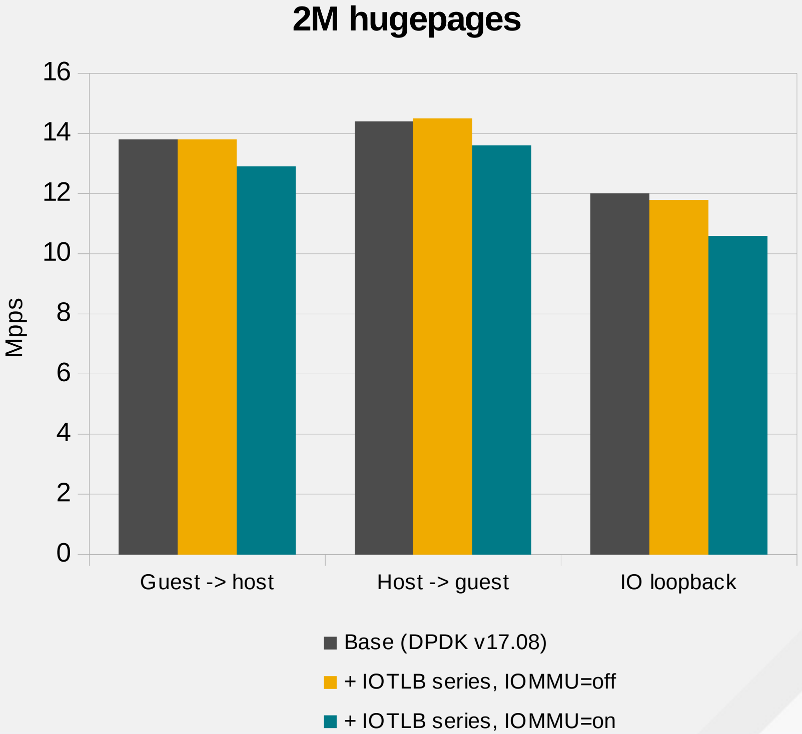


redhat.

Micro-benchmark using Testpmd

Benchmarks





Future improvements

Contiguous IOTLB entries merging

Future improvements

- Performance penalty with 2MB hugepages due to higher number of IOTLB entries

  → IOTLB cache lookup overhead
- Most of IOTLB entries are both virtually AND physically contiguous
- Rough prototype merging entries fixes performance penalty
  - Less IOTLB cache lookup iterations
  - Better CPU cache utilization
- Remaining questions:
  - Need to define invalidation strategy : invalidate all merged entry or split it?
  - Is there a performance impact with dynamic mappings?

Interval tree based IOTLB cache

Future improvements

- Vhost-kernel backend uses interval tree for its IOTLB cache implementation

  → O(log(n)) for lookup

- Current Vhost-user backend only implements sorted list

  → O(n) for lookup

- Required work

  → New interval tree lib in DPDK

  → Convert Vhost-user's IOTLB cache implementation

IOTLB misses batching

Future improvements

- IOMMU support with Virtio-net kernel driver not viable due to poor performance
  → Bursting broken due to IOTLB miss for every packets
- Before starting packets burst loop, translate all descriptors buffers addresses
  - If no missing translations, start the burst
  - If some, send IOTLB miss requests for all missing translations
- Might improve overall performance with multiple vhost-user ports per lcore

Conclusion

Conclusion

- Vhost-user design close to Vhost-kernel
- Reasonable performance impact with static mappings
  - And more improvements coming soon!
- Performance impact a blocker with dynamic mappings
- Special thanks to :
  - Jason Wang & Wei Xu – Vhost-kernel IOMMU support
  - Peter Xu – vIOMMU support in QEMU

redhat.

Questions?

THANK YOU!