Hardware Friendly Vhost vDPA
Towards an Efficient and Migratable Device Model

Si-Wei Liu, Oracle
Sr Principal Software Engineer

si-wei.liu@oracle.com

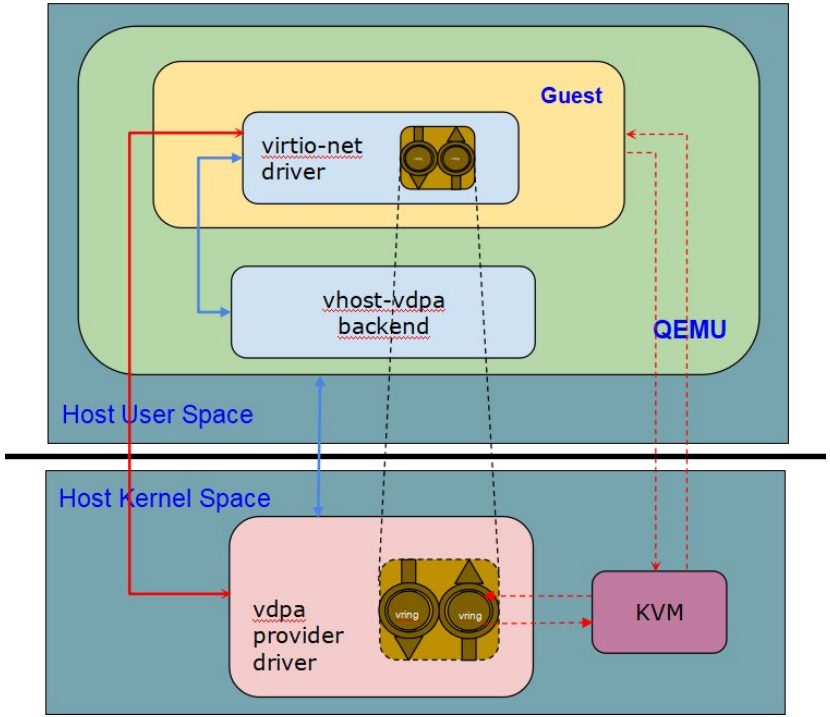
Agenda

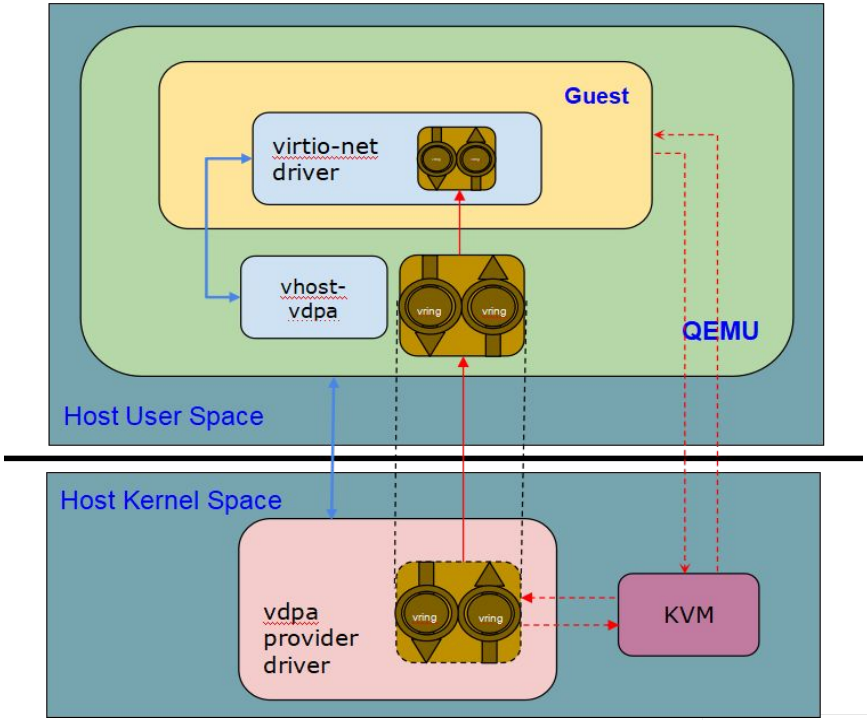
- Retrospect problems and challenges in Shadow VirtQueue implementation
- Propose new APIs for vDPA device state save and restore in device-neural and future compatible manner
- Expose guest invisible device state for resume via virtio-pci transport (nested guest)
- Update device model and migration stream to allow live migration between vhost-vdpa and QEMU's software VirtIO devices

Shadow VirtQueue overview

- Idea stemmed from DPDK's Software Assisted VDPA live migration design
- Emulate VirtQueue in the intermediate QEMU layer to intercept Used Ring access
- Interrupt and doorbell kick redirected to the Shadow VirtQueue in mediation
- Datapath Shadow VirtQueue active only during migration time
  - log dirty on behalf of underlying device
  - forward descriptors instead of buffers between guest and device
  - short blackout time during mode switch
- Control Shadow VirtQueue always active
  - keep close track of device state configured through Ctrl VQ on source
  - apply same set of device state through Ctrl VQ commands on migration destination
- Great emulation layer for realizing ring layout variants for e.g. virtio v0.9.5 legacy device emulation on top of v1.0 modern hardware

Shadow VirtQueue overview (cont.)





Shadow VirtQueue: Challenges

Inefficiency in restoring state on migration destination

- No resume support on virtio level, have to rely on device reset + reload vq state + replay cvq messages
- vDPA suspend uAPI was added, with no symmetric uAPI for resume
- Have to honor existing protocol and spec constrained order to apply command messages through ctrl vq
  - features bits are indicative but config space is not self-contained
  - ctrl vq wouldn't be ready until DRIVER_OK and qenable is set
  - on MQ net device, only single queue pair is enabled until _MQ_VQ_PAIRS_SET is received
  - out of spec trick to apply datapath enabling all at once, in order to work around some vendor's device due to massive time in repeatedly stopping and restarting datapath

1. User to create vdpa device with matching features
2. QEMU to set up config space (mac, mtu, link)
3. Configure data vqs (base, addr, size, …)
4. Configure and enable ctrl vq
5. QEMU to set DRIVER_OK and propagate to the backend driver
6. Backend driver receives _MAC_ADDR_SET via ctrl vq
7. Backend driver receives _MQ_PAIRS_SET via ctrl vq
8. Backend driver receives _MQ_RSS_CONFIG via ctrl vq

Shadow VirtQueue: Challenges (cont.)

Inefficiency in restoring state on migration destination

- The way of programming registers to configure virtqueue doesn't satisfy minimum downtime requirement with large number of queues, especially when comes to virtio based hardware device
- Device specific means, no common facility to restore device state
- Implication of DRIVER_OK and qenable: datapath is started while restoring state may run conflict with consistency and integrity
- No idempotency guaranteed when dealing with stateful device

Shadow VirtQueue: Challenges (cont.)

- QEMU Live Update requirement
  - Leverages Live Migration framework, but saves state in local file across exec'ing new QEMU binary
  - Open fds and kernel device states are preserved on exec
  - exec mode targets sub-second latency (a few hundred milliseconds of blackout) to pick up and recover all device's state
  - No shortcut exists for fast vhost-vdpa suspend/resume across live update reboot mode, have to repeat full cycle involving reset

Migration compatibility burden

- Device state for vhost-vdpa (for e.g netdev backend) is now modeled after QEMU's software emulated VirtIODevice
- In practice migration compatibility is not ruled by VirtIO spec but by QEMU's software implementation for virtio-net/blk/scsi/etc
- Generic vhost-vdpa[-pci] device can have its own self-contained VMSTATE, but would miss various features from software VirtIODevice
- Possible to migration between QEMU's virtio-net/blk/scsi emulated device and generic vhost-vdpa device?

Abstract device state in VirtIO standard

- Motivation to separate out VirtIO standard backed device state from QEMU's VirtIODevice implementation
- Ideally standard defined device state may consist of
  - generic device state (status, feature bits, num_queues)
  - per-virtqueue state (base, addr, size)
  - feature state present in config space (mac, mtu, link status, ...)
  - feature state invisible in config space (mac_table, vlan, rx, rss, ...)
  - device specific state not backed by feature (virtio-blk/scsi in-flight request)
- Bi-directional (backward and forward) migration compatibility needs to be assured

APIs for migrating vDPA device state

```
#define VHOST_BACKEND_F_MIGSTATE  0x5
#define VHOST_BACKEND_F_RESUME    0x6
#define VHOST_VDPA_RESUME         _IO(VHOST_VIRTIO, 0x7E)

struct vdpa_config_ops {
     /* Device ops */
     ...
     size_t (*get_mig_state_size)(struct vdpa_device *vdev);
     int (*save_mig_state)(struct vdpa_device *vdev, struct file *migf);
     int (*load_mig_state)(struct vdpa_device *vdev, struct file *migf);
     int (*resume)(struct vdpa_device *vdev);
     ...
};
```

On migration source

- **VHOST_VDPA_SUSPEND -> suspend()**
- **get_mig_state_size() to get the size of device state blob**
- **save_mig_state() to read in the byte stream for vdpa device state**
- **reset() - success, or resume() - failure**

On migration destination

- **reset()**
- **VHOST_VDPA_SUSPEND -> suspend()**
- **load_mig_state() to restore vdpa device state from the byte stream**
- **VHOST_VDPA_RESUME -> resume()**

vDPA device state breakdown

```
#define VIRTIO_MIG_STATE_TYPE_DEVICE    0
#define VIRTIO_MIG_STATE_TYPE_VQ        1
#define VIRTIO_MIG_STATE_TYPE_CONFIG    2
#define VIRTIO_MIG_STATE_TYPE_FEATURE   3
#define VIRTIO_MIG_STATE_TYPE_PLATFORM  4
#define VIRTIO_MIG_STATE_TYPE_VENDOR    255

struct vdpa_mig_state {
    struct virtio_mig_dev_common_state dev_state;
    struct virtio_mig_vq_state vq_state;
    struct virtio_mig_config_state cfg_state;
    struct virtio_mig_feat_state feat_state; [optional]
    [other optional states followed on ...]
};
```

VirtIO state - device common

```
struct virtio_mig_state_header {
     le32 type;
     le32 len;
};

struct virtio_mig_dev_common_state {
     struct virtio_mig_state_header hdr;
     struct virtio_mig_dev_common_data data;
};

struct virtio_mig_dev_common_data {
     le32 vendor_id;
     le32 device_id;
     le32 generation;
     le64 device_features;
     le64 driver_features;
     u8 status;
};
```

```
#define VIRTIO_MIG_STATE_TYPE_DEVICE   0

#define VIRTIO_MIG_DEVICE_T_COMMON     0
#define VIRTIO_MIG_DEVICE_T_NET        1
#define VIRTIO_MIG_DEVICE_T_BLK        2
#define VIRTIO_MIG_DEVICE_T_SCSI       8
```

```
hdr.type = VIRTIO_MIG_STATE_TYPE_DEVICE << 24 |
           VIRTIO_MIG_DEVICE_T_COMMON;
hdr.len  = sizeof (struct virtio_mig_dev_common_data);
```

VirtIO state - device specific

```
struct virtio_mig_state_header {
     le32 type;
     le32 len;
};

struct virtio_mig_dev_blk_state {
     struct virtio_mig_state_header hdr;
     struct virtio_mig_dev_blk_data data;
};

struct virtio_mig_dev_blk_data {
     struct virtio_mig_blk_inflight infl;
     ...
};

struct virtio_mig_blk_inflight {
     le64 num_inflight;
     le16 qsize;
     le64 addr[num_inflight];
};
```

```
#define VIRTIO_MIG_STATE_TYPE_DEVICE   0

#define VIRTIO_MIG_DEVICE_T_BLK        2
```

```
hdr.type = VIRTIO_MIG_STATE_TYPE_DEVICE << 24 |
           VIRTIO_MIG_DEVICE_T_BLK;
hdr.len  = sizeof (struct virtio_mig_dev_blk_data);
```

VirtIO state - virtqueue

```
struct virtio_mig_vq_state {
     struct virtio_mig_state_header hdr;
     le16 num_queues;
     struct virtio_mig_per_vq_data vq_state[];
};

struct virtio_mig_per_vq_data {
     le32 qsize;
     u8 qenabled;
     le64 desc;
     le64 avail;
     le64 used;
     union {
          struct virtio_mig_vq_split_state
split;
          struct virtio_mig_vq_packed_state
packed;
     };
};
```

```
#define VIRTIO_MIG_STATE_TYPE_VQ   1

#define VIRTIO_MIG_VQ_T_COMMON     0
```

```
hdr.type = VIRTIO_MIG_STATE_TYPE_VQ << 24 |
           VIRTIO_MIG_VQ_T_COMMON;
hdr.len  = sizeof (num_queues) +
           sizeof (struct virtio_mig_per_vq_data) *
           num_queues;
```

```
struct virtio_mig_vq_split_state {
     le16 avail_index;
     le16 used_index;
};

struct virtio_mig_vq_packed_state {
     le16 avail_wrapped:1;
     le16 avail_index:15;
     le16 used_wrapped:1;
     le16 used_index:15;
};
```

VirtIO state - config space

```
struct virtio_mig_config_state {
     struct virtio_mig_state_header hdr;
     union {
          struct virtio_net_config net;
          struct virtio_blk_config blk;
          struct virtio_scsi_config scsi;
          ...
     };
};

struct virtio_net_config {
        u8 mac[6];
        le16 status;
        le16 max_virtqueue_pairs;
        le16 mtu;
        le32 speed;
        u8 duplex;
        u8 rss_max_key_size;
        le16 rss_max_indirection_table_length;
        le32 supported_hash_types;
};
```

```
#define VIRTIO_MIG_STATE_TYPE_CONFIG   2

#define VIRTIO_MIG_CONFIG_T_NET        1
#define VIRTIO_MIG_CONFIG_T_BLK        2
#define VIRTIO_MIG_CONFIG_T_SCSI       8
```

```
hdr.type = VIRTIO_MIG_STATE_TYPE_CONFIG << 24 |
           VIRTIO_MIG_CONFIG_T_NET;
hdr.len  = sizeof (struct virtio_net_config);
```

virtio-net state - guest invisible config

```
struct virtio_mig_feat_state {
     union {
          struct virtio_mig_cfg_net_features net;
          struct virtio_mig_cfg_blk_features blk;
          struct virtio_mig_cfg_scsi_features scsi;
          ...
     };
};
```

```
hdr.type = VIRTIO_MIG_STATE_TYPE_FEATURE << 24 |
           feature << 16 |
           class << 8 |
           command;
hdr.len  = sizeof
           (struct virtio_mig_cfg_xxx_features);
```

```
#define VIRTIO_MIG_STATE_TYPE_FEATURE 3

#define VIRTIO_MIG_CONFIG_T_NET       1
#define VIRTIO_MIG_CONFIG_T_BLK       2
#define VIRTIO_MIG_CONFIG_T_SCSI      8
```

- header's type encoding for guest invisible auxiliary config is device specific, and feature based
- feature derived from virtio feature bit, but may include sub-feature, e.g. class, command for virtio-net's ctrl virtqueue
- struct casting depends on which device type present in the config space state, e.g. `VIRTIO_MIG_CONFIG_T_NET`

virtio-net state - guest offloads

```
hdr.type = VIRTIO_MIG_STATE_TYPE_FEATURE << 24 |
           VIRTIO_NET_F_CTRL_GUEST_OFFLOADS << 16 |
           VIRTIO_NET_CTRL_GUEST_OFFLOADS << 8 |
           VIRTIO_NET_CTRL_GUEST_OFFLOADS_SET;
hdr.len  = sizeof (struct virtio_mig_..._offloads) -
           sizeof (struct virtio_mig_state_header);
```

```
#define VIRTIO_NET_F_CTRL_GUEST_OFFLOADS      2
#define VIRTIO_NET_CTRL_GUEST_OFFLOADS        5
#define VIRTIO_NET_CTRL_GUEST_OFFLOADS_SET    0

#define VIRTIO_NET_F_GUEST_CSUM       1
#define VIRTIO_NET_F_GUEST_TSO4       7
#define VIRTIO_NET_F_GUEST_TSO6       8
#define VIRTIO_NET_F_GUEST_ECN        9
#define VIRTIO_NET_F_GUEST_UFO        10
```

```
struct virtio_mig_cfg_net_features {
      struct virtio_mig_state_header hdr;
      struct virtio_mig_cfg_net_data data;
};
struct virtio_mig_cfg_net_data {
      le32 nfeatures;
      struct virtio_mig_cfg_net_ctrl_guest_offloads offloads;
      struct virtio_mig_cfg_net_ctrl_mq_vq_pairs mq_pairs;
      ...
      struct virtio_mig_cfg_net_ctrl_mac_table mac_table;
      struct virtio_mig_cfg_net_ctrl_vlan vlan_table;
      ...
};
```

```
struct virtio_mig_cfg_net_ctrl_guest_offloads {
      struct virtio_mig_state_header hdr;
      le64 offloads;
      le64 reserved;
};
```

virtio-net state - multiqueue

```
hdr.type = VIRTIO_MIG_STATE_TYPE_FEATURE << 24 |
           VIRTIO_NET_F_MQ << 16 |
           VIRTIO_NET_CTRL_MQ << 8 |
           VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET;
hdr.len  = sizeof (struct virtio_mig_..._vq_pairs) -
           sizeof (struct virtio_mig_state_header);
```

```
#define VIRTIO_NET_F_MQ              22
#define VIRTIO_NET_CTRL_MQ           4
#define VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET 0
```

```
struct virtio_mig_cfg_net_features {
      struct virtio_mig_state_header hdr;
      struct virtio_mig_cfg_net_data data;
};
struct virtio_mig_cfg_net_data {
      le32 nfeatures;
      struct virtio_mig_cfg_net_ctrl_guest_offloads offloads;
      struct virtio_mig_cfg_net_ctrl_mq_vq_pairs mq_pairs;
      ...
      struct virtio_mig_cfg_net_ctrl_mac_table mac_table;
      struct virtio_mig_cfg_net_ctrl_vlan vlan_table;
      ...
};
```

```
struct virtio_mig_cfg_net_ctrl_mq_vq_pairs {
      struct virtio_mig_state_header hdr;
      le16 cur_virtqueue_pairs;
};
```

THE LINUX FOUNDATION

virtio-net state - MAC table

```
hdr.type = VIRTIO_MIG_STATE_TYPE_FEATURE << 24 |
           VIRTIO_NET_F_CTRL_MAC_ADDR << 16 |
           VIRTIO_NET_CTRL_MAC << 8 |
           VIRTIO_NET_CTRL_MAC_TABLE_SET;
hdr.len  = sizeof (struct virtio_mig_..._mac_table) -
           sizeof (struct virtio_mig_state_header);
```

```
#define VIRTIO_NET_F_CTRL_MAC_ADDR      23
#define VIRTIO_NET_CTRL_MAC             1
#define VIRTIO_NET_CTRL_MAC_TABLE_SET   0
```

```
struct virtio_mig_cfg_net_features {
      struct virtio_mig_state_header hdr;
      struct virtio_mig_cfg_net_data data;
};
struct virtio_mig_cfg_net_data {
      le32 nfeatures;
      struct virtio_mig_cfg_net_ctrl_guest_offloads offloads;
      struct virtio_mig_cfg_net_ctrl_mq_vq_pairs mq_pairs;
      ...
      struct virtio_mig_cfg_net_ctrl_mac_table mac_table;
      struct virtio_mig_cfg_net_ctrl_vlan vlan_table;
      ...
};
```

```
struct virtio_mig_cfg_net_ctrl_mac_table {
      struct virtio_mig_state_header hdr;
      le16 num_unicast;
      u8 unicast_macs[num_unicast][6];
      le16 num_multicast;
      u8 multicast_macs[num_multicast][6];
};
```

THE LINUX FOUNDATION

virtio-net state - VLAN table

```
hdr.type = VIRTIO_MIG_STATE_TYPE_FEATURE << 24 |
           VIRTIO_NET_F_CTRL_VLAN << 16 |
           VIRTIO_NET_CTRL_VLAN << 8 |
           VIRTIO_NET_CTRL_VLAN_ADD;
hdr.len  = sizeof (struct virtio_mig_..._ctrl_vlan) -
           sizeof (struct virtio_mig_state_header);
```

```
#define VIRTIO_NET_F_CTRL_VLAN      19
#define VIRTIO_NET_CTRL_VLAN        2
#define VIRTIO_NET_CTRL_VLAN_ADD    0
```

```
struct virtio_mig_cfg_net_features {
      struct virtio_mig_state_header hdr;
      struct virtio_mig_cfg_net_data data;
};
struct virtio_mig_cfg_net_data {
      le32 nfeatures;
      struct virtio_mig_cfg_net_ctrl_guest_offloads offloads;
      struct virtio_mig_cfg_net_ctrl_mq_vq_pairs mq_pairs;
      ...
      struct virtio_mig_cfg_net_ctrl_mac_table mac_table;
      struct virtio_mig_cfg_net_ctrl_vlan vlan_table;
      ...
};
```

```
struct virtio_mig_cfg_net_ctrl_vlan {
      struct virtio_mig_state_header hdr;
      le32 vlans[128];
};
```

Auxiliary device state PCI capability

```
struct virtio_mig_feat_state {
      union {
            struct virtio_mig_cfg_net_features net;
            struct virtio_mig_cfg_blk_features blk;
            struct virtio_mig_cfg_scsi_features scsi;
            ...
      };
};

struct virtio_mig_cfg_net_features {
      struct virtio_mig_state_header hdr;
      struct virtio_mig_cfg_net_data data;
};
struct virtio_mig_cfg_net_data {
      le32 nfeatures;
      struct virtio_mig_cfg_net_ctrl_guest_offloads offloads;
      struct virtio_mig_cfg_net_ctrl_mq_vq_pairs mq_pairs;
      ...
      struct virtio_mig_cfg_net_ctrl_mac_table mac_table;
      struct virtio_mig_cfg_net_ctrl_vlan vlan_table;
      ...
};
```

```
#define VIRTIO_PCI_CAP_AUX_STATE_CFG    13

struct virtio_pci_aux_state_cap {
     struct virtio_pci_cap cap;
     struct virtio_mig_feat_state auxcfg;
};
```

- Present on MMIO BAR on VirtIO PCI transport
- No direct exposure to nested L2 guest
- Only valid for read and write in suspended state from L1, after device stops
- Can be extended to other transport, e.g. Admin Queue or Transport VirtQueue

Backward compatibility with software VirtIO

- Device state in the blob of `struct vdpa_mig_state` is interchangeable to the corresponding `VirtIODevice`'s `VMStateDescription` in QEMU's migration stream
- Adapter layer to translate `struct vdpa_mig_state` from/to `VMStateDescription` is needed for backward compatibility (fallback when either side of migration runs older QEMU)
- Newer QEMU may transfer the standard derived device state in the `struct vdpa_mig_state` blob as-is to migration stream, when vDPA backend is capable

Compatibility between vhost-vdpa versions

- Admin user's job to create vdpa with matched features as in the source on migration destination
- State structure in `vdpa_mig_state` stream may grow, but should expose a flag to vdpa admin tool for user's identification
- Incompatible addition to existing feature or deprecated feature in favor of a new one should come up with new device state type to define
- Destination may assume safe to resume when it comes to a previous version of structure with shorter length in the `vdpa_mig_state` stream
- Destination should fail to resume if seeing an unknown type of structure, or if coming to a structure with longer length than its own from the `vdpa_mig_state` stream

Shadow VirtQueue v.s. Resume-able

	Shadow VQ	Resume-able vDPA
Availability	Works with existing spec and vhost protocol Stateful device doesn't apply	Spec extension & new API needed Can work with stateful device
Hardware/vendor driver support	Works with any hardware Does not need driver support	Needed
Datapath Performance	Mediation Slight impact may exist	Passthrough Zero overhead
Datapath Switchover	Blackout time exists	0 ms
Ring Layout Translation	Possible	Not Applicable
CPU Usage	Moderate	0%
Dirty Tracking	Inherent	Platform IOMMU support needed, or device assisted (*)

Q & A

Thanks!

KVM FORUM

Device assisted tracking - Dirty Bitmap

- Dirty Bitmap mostly suitable for para-virtual device
  - Not possible to update just a bit of memory in PCI write
  - Real hardware device needs to get a full cycle of Read-Modify-Write to update a bit
  - Efficiency problem in PCI bus and cache synchronization cost in CCIX fabric
- Bytemap variant to avoid full Read-Modify-Write cycle
  - But would take 8x memory, and bad cache usage
  - May take some optimization to limit to the range device DMA can get to

Device assisted tracking - Dirty Ring

- Borrowed from the idea of KVM dirty ring https://lwn.net/Articles/833784/
- Bitmap or Bytemap takes up substantial amount of host memory for dirty tracking
  - ✔ grows linearly with the size of guest memory
  - ✔ space and time inefficiency, particularly for larger guest e.g. with TB+ of memory
- Extra benefit: intrinsic throttling and convergence
  - ✔ Ring full condition automatically throttles and rate limits I/O processing
- QEMU needs to move to ring based dirty tracking for best efficiency and performance
- Should be a better tracking facility than bitmap for general use cases
- Some specific cases Ring might not play well against Bitmap or Bytemap
  - ✔ Small VM but intensive I/O dirty rate
  - ✔ Limited number of Dirty Ring entries causing significant I/O performance degradation
  - ✔ Lack of host compute power that stalls I/O processing too often
- In need of PoC and real world data to verify theory and quantify migration-time performance