VFIO - "Virtual Function I/O" [1]

Many modern systems now provide DMA and interrupt remapping facilities to help ensure I/O devices behave within the boundaries they've been allotted. This includes x86 hardware with AMD-Vi and Intel VT-d, POWER systems with Partitionable Endpoints (PEs) and embedded PowerPC systems such as Freescale PAMU. The VFIO driver is an IOMMU/device agnostic framework for exposing direct device access to userspace, in a secure, IOMMU protected environment. In other words, this allows safe [2], non-privileged, userspace drivers.

Why do we want that? Virtual machines often make use of direct device access ("device assignment") when configured for the highest possible I/O performance. From a device and host perspective, this simply turns the VM into a userspace driver, with the benefits of significantly reduced latency, higher bandwidth, and direct use of bare-metal device drivers [3].

Some applications, particularly in the high performance computing field, also benefit from low-overhead, direct device access from userspace. Examples include network adapters (often non-TCP/IP based) and compute accelerators. Prior to VFIO, these drivers had to either go through the full development cycle to become proper upstream driver, be maintained out of tree, or make use of the UIO framework, which has no notion of IOMMU protection, limited interrupt support, and requires root privileges to access things like PCI configuration space.

The VFIO driver framework intends to unify these, replacing both the KVM PCI specific device assignment code as well as provide a more secure, more featureful userspace driver environment than UIO.

Groups, Devices, and IOMMUs

Devices are the main target of any I/O driver. Devices typically create a programming interface made up of I/O access, interrupts, and DMA. Without going into the details of each of these, DMA is by far the most critical aspect for maintaining a secure environment as allowing a device read-write access to system memory imposes the greatest risk to the overall system integrity.

To help mitigate this risk, many modern IOMMUs now incorporate isolation properties into what was, in many cases, an interface only meant for translation (ie. solving the addressing problems of devices with limited address spaces). With this, devices can now be isolated from each other and from arbitrary memory access, thus allowing things like secure direct assignment of devices into virtual machines.

This isolation is not always at the granularity of a single device though. Even when an IOMMU is capable of this, properties of devices, interconnects, and IOMMU topologies can each reduce this isolation. For instance, an individual device may be part of a larger multi- function enclosure. While the IOMMU may be able to distinguish between devices within the enclosure, the enclosure may not require transactions between devices to reach the IOMMU. Examples of this could be anything from a multi-function PCI device with backdoors between functions to a non-PCI-ACS (Access Control Services) capable bridge allowing redirection without reaching the IOMMU. Topology can also play a factor in terms of hiding devices. A PCIe-to-PCI bridge masks the devices behind it, making transaction appear as if from the bridge itself. Obviously IOMMU design plays a major factor as well.

Therefore, while for the most part an IOMMU may have device level granularity, any system is susceptible to reduced granularity. The IOMMU API therefore supports a notion of IOMMU groups. A group is a set of devices which is isolatable from all other devices in the system. Groups are therefore the unit of ownership used by VFIO.

While the group is the minimum granularity that must be used to ensure secure user access, it's not necessarily the preferred granularity. In IOMMUs which make use of page tables, it may be possible to share a set of page tables between different groups, reducing the overhead both to the platform (reduced TLB thrashing, reduced duplicate page tables), and to the user (programming only a single set of translations). For this reason, VFIO makes use of a container class, which may hold one or more groups. A container is created by simply opening the /dev/vfio/vfio character device.

On its own, the container provides little functionality, with all but a couple version and extension query interfaces locked away. The user needs to add a group into the container for the next level of functionality. To do this, the user first needs to identify the group associated with the desired device. This can be done using the sysfs links described in the example below. By unbinding the device from the host driver and binding it to a VFIO driver, a new VFIO group will appear for the group as /dev/vfio/$GROUP, where $GROUP is the IOMMU group number of which the device is a member. If the IOMMU group contains multiple devices, each will need to be bound to a VFIO driver before operations on the VFIO group are allowed (it's also sufficient to only unbind the device from host drivers if a VFIO driver is unavailable; this will make the group available, but not that particular device). TBD - interface for disabling driver probing/locking a device.

Once the group is ready, it may be added to the container by opening the VFIO group character device (/dev/vfio/$GROUP) and using the VFIO_GROUP_SET_CONTAINER ioctl, passing the file descriptor of the previously opened container file. If desired and if the IOMMU driver supports sharing the IOMMU context between groups, multiple groups may be set to the same container. If a group fails to set to a container with existing groups, a new empty container will need to be used instead.

With a group (or groups) attached to a container, the remaining ioctls become available, enabling access to the VFIO IOMMU interfaces. Additionally, it now becomes possible to get file descriptors for each device within a group using an ioctl on the VFIO group file descriptor.

The VFIO device API includes ioctls for describing the device, the I/O regions and their read/write/mmap offsets on the device descriptor, as well as mechanisms for describing and registering interrupt notifications.

VFIO Usage Example

Assume user wants to access PCI device 0000:06:0d.0:

```
$ readlink /sys/bus/pci/devices/0000:06:0d.0/iommu_group
../../../../kernel/iommu_groups/26
```

This device is therefore in IOMMU group 26. This device is on the pci bus, therefore the user will make use of vfio-pci to manage the group:

```
# modprobe vfio-pci
```

Binding this device to the vfio-pci driver creates the VFIO group character devices for this group:

```
$ lspci -n -s 0000:06:0d.0
06:0d.0 0401: 1102:0002 (rev 08)
# echo 0000:06:0d.0 > /sys/bus/pci/devices/0000:06:0d.0/driver/unbind
# echo 1102 0002 > /sys/bus/pci/drivers/vfio-pci/new_id
```

Now we need to look at what other devices are in the group to free it for use by VFIO:

```
$ ls -l /sys/bus/pci/devices/0000:06:0d.0/iommu_group/devices
total 0
lrwxrwxrwx. 1 root root 0 Apr 23 16:13 0000:00:1e.0 ->
        ../../../../devices/pci0000:00/0000:00:1e.0
lrwxrwxrwx. 1 root root 0 Apr 23 16:13 0000:06:0d.0 ->
        ../../../../devices/pci0000:00/0000:00:1e.0/0000:06:0d.0
lrwxrwxrwx. 1 root root 0 Apr 23 16:13 0000:06:0d.1 ->
        ../../../../devices/pci0000:00/0000:00:1e.0/0000:06:0d.1
```

This device is behind a PCIe-to-PCI bridge [4], therefore we also need to add device 0000:06:0d.1 to the group following the same procedure as above. Device 0000:00:1e.0 is a bridge that does not currently have a host driver, therefore it's not required to bind this device to the vfio-pci driver (vfio-pci does not currently support PCI bridges).

The final step is to provide the user with access to the group if unprivileged operation is desired (note that /dev/vfio/vfio provides no capabilities on its own and is therefore expected to be set to mode 0666 by the system):

```
# chown user:user /dev/vfio/26
```

The user now has full access to all the devices and the iommu for this group and can access them as follows:

```
int container, group, device, i;
struct vfio_group_status group_status =
                                { .argsz = sizeof(group_status) };
struct vfio_iommu_type1_info iommu_info = { .argsz = sizeof(iommu_info) };
struct vfio_iommu_type1_dma_map dma_map = { .argsz = sizeof(dma_map) };
struct vfio_device_info device_info = { .argsz = sizeof(device_info) };

/* Create a new container */
container = open("/dev/vfio/vfio", O_RDWR);

if (ioctl(container, VFIO_GET_API_VERSION) != VFIO_API_VERSION)
        /* Unknown API version */

if (!ioctl(container, VFIO_CHECK_EXTENSION, VFIO_TYPE1_IOMMU))
        /* Doesn't support the IOMMU driver we want. */

/* Open the group */
group = open("/dev/vfio/26", O_RDWR);

/* Test the group is viable and available */
ioctl(group, VFIO_GROUP_GET_STATUS, &group_status);

if (!(group_status.flags & VFIO_GROUP_FLAGS_VIABLE))
        /* Group is not viable (ie, not all devices bound for vfio) */

/* Add the group to the container */
ioctl(group, VFIO_GROUP_SET_CONTAINER, &container);

/* Enable the IOMMU model we want */
ioctl(container, VFIO_SET_IOMMU, VFIO_TYPE1_IOMMU);

/* Get addition IOMMU info */
ioctl(container, VFIO_IOMMU_GET_INFO, &iommu_info);

/* Allocate some space and setup a DMA mapping */
dma_map.vaddr = mmap(0, 1024 * 1024, PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
dma_map.size = 1024 * 1024;
```

```
dma_map.iova = 0; /* 1MB starting at 0x0 from device view */
dma_map.flags = VFIO_DMA_MAP_FLAG_READ | VFIO_DMA_MAP_FLAG_WRITE;

ioctl(container, VFIO_IOMMU_MAP_DMA, &dma_map);

/* Get a file descriptor for the device */
device = ioctl(group, VFIO_GROUP_GET_DEVICE_FD, "0000:06:0d.0");

/* Test and setup the device */
ioctl(device, VFIO_DEVICE_GET_INFO, &device_info);

for (i = 0; i < device_info.num_regions; i++) {
        struct vfio_region_info reg = { .argsz = sizeof(reg) };

        reg.index = i;

        ioctl(device, VFIO_DEVICE_GET_REGION_INFO, &reg);

        /* Setup mappings... read/write offsets, mmaps
         * For PCI devices, config space is a region */
}

for (i = 0; i < device_info.num_irqs; i++) {
        struct vfio_irq_info irq = { .argsz = sizeof(irq) };

        irq.index = i;

        ioctl(device, VFIO_DEVICE_GET_IRQ_INFO, &irq);

        /* Setup IRQs... eventfds, VFIO_DEVICE_SET_IRQS */
}

/* Gratuitous device reset and go... */
ioctl(device, VFIO_DEVICE_RESET);
```

VFIO User API

Please see include/linux/vfio.h for complete API documentation.

VFIO bus driver API

VFIO bus drivers, such as vfio-pci make use of only a few interfaces into VFIO core. When devices are bound and unbound to the driver, the driver should call vfio_register_group_dev() and vfio_unregister_group_dev() respectively:

```
void vfio_init_group_dev(struct vfio_device *device,
                         struct device *dev,
                         const struct vfio_device_ops *ops);
void vfio_uninit_group_dev(struct vfio_device *device);
int vfio_register_group_dev(struct vfio_device *device);
void vfio_unregister_group_dev(struct vfio_device *device);
```

The driver should embed the vfio_device in its own structure and call vfio_init_group_dev() to pre-configure it before going to registration and call vfio_uninit_group_dev() after completing the un-registration. vfio_register_group_dev() indicates to the core to begin tracking the iommu_group of the specified dev and register the dev as owned by a VFIO bus driver. Once vfio_register_group_dev() returns it is possible for userspace to start accessing the driver, thus the driver should ensure it is completely ready before calling it. The driver provides an ops structure for callbacks similar to a file operations structure:

```
struct vfio_device_ops {
        int     (*open)(struct vfio_device *vdev);
        void    (*release)(struct vfio_device *vdev);
        ssize_t (*read)(struct vfio_device *vdev, char __user *buf,
                        size_t count, loff_t *ppos);
        ssize_t (*write)(struct vfio_device *vdev,
                         const char __user *buf,
                         size_t size, loff_t *ppos);
        long    (*ioctl)(struct vfio_device *vdev, unsigned int cmd,
                         unsigned long arg);
        int     (*mmap)(struct vfio_device *vdev,
                        struct vm_area_struct *vma);
};
```

Each function is passed the vdev that was originally registered in the vfio_register_group_dev() call above. This allows the bus driver to obtain its private data using container_of(). The open/release callbacks are issued when a new file descriptor is created for a device (via VFIO_GROUP_GET_DEVICE_FD). The ioctl interface provides a direct pass through for VFIO_DEVICE_* ioctls. The read/write/mmap interfaces implement the device region access defined by the device's own VFIO_DEVICE_GET_REGION_INFO ioctl.

PPC64 sPAPR implementation note

This implementation has some specifics:

1. On older systems (POWER7 with P5IOC2/IODA1) only one IOMMU group per container is supported as an IOMMU table is allocated at the boot time, one table per a IOMMU group which is a Partitionable Endpoint (PE) (PE is often a PCI domain but not always).

   Newer systems (POWER8 with IODA2) have improved hardware design which allows to remove this limitation and have multiple IOMMU groups per a VFIO container.

2. The hardware supports so called DMA windows - the PCI address range within which DMA transfer is allowed, any attempt to access address space out of the window leads to the whole PE isolation.

3. PPC64 guests are paravirtualized but not fully emulated. There is an API to map/unmap pages for DMA, and it normally maps 1..32 pages per call and currently there is no way to reduce the number of calls. In order to make things faster, the map/unmap handling has been implemented in real mode which provides an excellent performance which has limitations such as inability to do locked pages accounting in real time.

4. According to sPAPR specification, A Partitionable Endpoint (PE) is an I/O subtree that can be treated as a unit for the purposes of partitioning and error recovery. A PE may be a single or multi-function IOA (IO Adapter), a function of a multi-function IOA, or multiple IOAs (possibly including switch and bridge structures above the multiple IOAs). PPC64 guests detect PCI errors and recover from them via EEH RTAS services, which works on the basis of additional ioctl commands.

   So 4 additional ioctls have been added:

   VFIO_IOMMU_SPAPR_TCE_GET_INFO
   : returns the size and the start of the DMA window on the PCI bus.

   VFIO_IOMMU_ENABLE
   : enables the container. The locked pages accounting is done at this point. This lets user first to know what the DMA window is and adjust rlimit before doing any real job.

   VFIO_IOMMU_DISABLE
   : disables the container.

   VFIO_EEH_PE_OP
   : provides an API for EEH setup, error detection and recovery.

   The code flow from the example above should be slightly changed:

```
struct vfio_eeh_pe_op pe_op = { .argsz = sizeof(pe_op), .flags = 0 };

.....
/* Add the group to the container */
ioctl(group, VFIO_GROUP_SET_CONTAINER, &container);

/* Enable the IOMMU model we want */
ioctl(container, VFIO_SET_IOMMU, VFIO_SPAPR_TCE_IOMMU)

/* Get addition sPAPR IOMMU info */
vfio_iommu_spapr_tce_info spapr_iommu_info;
ioctl(container, VFIO_IOMMU_SPAPR_TCE_GET_INFO, &spapr_iommu_info);

if (ioctl(container, VFIO_IOMMU_ENABLE))
        /* Cannot enable container, may be low rlimit */

/* Allocate some space and setup a DMA mapping */
dma_map.vaddr = mmap(0, 1024 * 1024, PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);

dma_map.size = 1024 * 1024;
dma_map.iova = 0; /* 1MB starting at 0x0 from device view */
dma_map.flags = VFIO_DMA_MAP_FLAG_READ | VFIO_DMA_MAP_FLAG_WRITE;

/* Check here is .iova/.size are within DMA window from spapr_iommu_info */
ioctl(container, VFIO_IOMMU_MAP_DMA, &dma_map);

/* Get a file descriptor for the device */
device = ioctl(group, VFIO_GROUP_GET_DEVICE_FD, "0000:06:0d.0");

....

/* Gratuitous device reset and go... */
ioctl(device, VFIO_DEVICE_RESET);

/* Make sure EEH is supported */
ioctl(container, VFIO_CHECK_EXTENSION, VFIO_EEH);
```

```
/* Enable the EEH functionality on the device */
pe_op.op = VFIO_EEH_PE_ENABLE;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/* You're suggested to create additional data struct to represent
 * PE, and put child devices belonging to same IOMMU group to the
 * PE instance for later reference.
 */

/* Check the PE's state and make sure it's in functional state */
pe_op.op = VFIO_EEH_PE_GET_STATE;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/* Save device state using pci_save_state().
 * EEH should be enabled on the specified device.
 */

....

/* Inject EEH error, which is expected to be caused by 32-bits
 * config load.
 */
pe_op.op = VFIO_EEH_PE_INJECT_ERR;
pe_op.err.type = EEH_ERR_TYPE_32;
pe_op.err.func = EEH_ERR_FUNC_LD_CFG_ADDR;
pe_op.err.addr = 0ul;
pe_op.err.mask = 0ul;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

....

/* When 0xFF's returned from reading PCI config space or IO BARs
 * of the PCI device. Check the PE's state to see if that has been
 * frozen.
 */
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/* Waiting for pending PCI transactions to be completed and don't
 * produce any more PCI traffic from/to the affected PE until
 * recovery is finished.
 */

/* Enable IO for the affected PE and collect logs. Usually, the
 * standard part of PCI config space, AER registers are dumped
 * as logs for further analysis.
 */
pe_op.op = VFIO_EEH_PE_UNFREEZE_IO;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/*
 * Issue PE reset: hot or fundamental reset. Usually, hot reset
 * is enough. However, the firmware of some PCI adapters would
 * require fundamental reset.
 */
pe_op.op = VFIO_EEH_PE_RESET_HOT;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);
pe_op.op = VFIO_EEH_PE_RESET_DEACTIVATE;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/* Configure the PCI bridges for the affected PE */
pe_op.op = VFIO_EEH_PE_CONFIGURE;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/* Restored state we saved at initialization time. pci_restore_state()
 * is good enough as an example.
 */

/* Hopefully, error is recovered successfully. Now, you can resume to
 * start PCI traffic to/from the affected PE.
 */

....
```

5. There is v2 of SPAPR TCE IOMMU. It deprecates VFIO_IOMMU_ENABLE/ VFIO_IOMMU_DISABLE and implements 2 new ioctls: VFIO_IOMMU_SPAPR_REGISTER_MEMORY and VFIO_IOMMU_SPAPR_UNREGISTER_MEMORY (which are unsupported in v1 IOMMU).

   PPC64 paravirtualized guests generate a lot of map/unmap requests, and the handling of those includes pinning/unpinning pages and updating mm:locked_vm counter to make sure we do not exceed the rlimit. The v2 IOMMU splits accounting and pinning into separate operations:

- VFIO_IOMMU_SPAPR_REGISTER_MEMORY/VFIO_IOMMU_SPAPR_UNREGISTER_MEMORY ioctls receive a user space address and size of the block to be pinned. Bisecting is not supported and VFIO_IOMMU_UNREGISTER_MEMORY is expected to be called with the exact address and size used for registering the memory block. The userspace is not expected to call these often. The ranges are stored in a linked list in a VFIO container.
- VFIO_IOMMU_MAP_DMA/VFIO_IOMMU_UNMAP_DMA ioctls only update the actual IOMMU table and do not do pinning; instead these check that the userspace address is from pre-registered range.

This separation helps in optimizing DMA for guests.

6. sPAPR specification allows guests to have an additional DMA window(s) on a PCI bus with a variable page size. Two ioctls have been added to support this: VFIO_IOMMU_SPAPR_TCE_CREATE and VFIO_IOMMU_SPAPR_TCE_REMOVE. The platform has to support the functionality or error will be returned to the userspace. The existing hardware supports up to 2 DMA windows, one is 2GB long, uses 4K pages and called "default 32bit window"; the other can be as big as entire RAM, use different page size, it is optional - guests create those in run-time if the guest driver supports 64bit DMA.

   VFIO_IOMMU_SPAPR_TCE_CREATE receives a page shift, a DMA window size and a number of TCE table levels (if a TCE table is going to be big enough and the kernel may not be able to allocate enough of physically contiguous memory). It creates a new window in the available slot and returns the bus address where the new window starts. Due to hardware limitation, the user space cannot choose the location of DMA windows.

   VFIO_IOMMU_SPAPR_TCE_REMOVE receives the bus start address of the window and removes it.

---

[1] VFIO was originally an acronym for "Virtual Function I/O" in its initial implementation by Tom Lyon while as Cisco. We've since outgrown the acronym, but it's catchy.

[2] "safe" also depends upon a device being "well behaved". It's possible for multi-function devices to have backdoors between functions and even for single function devices to have alternative access to things like PCI config space through MMIO registers. To guard against the former we can include additional precautions in the IOMMU driver to group multi-function PCI devices together (iommu=group_mf). The latter we can't prevent, but the IOMMU should still provide isolation. For PCI, SR-IOV Virtual Functions are the best indicator of "well behaved", as these are designed for virtualization usage models.

[3] As always there are trade-offs to virtual machine device assignment that are beyond the scope of VFIO. It's expected that future IOMMU technologies will reduce some, but maybe not all, of these trade-offs.

[4] In this case the device is below a PCI bridge, so transactions from either function of the device are indistinguishable to the iommu:

```
-[0000:00]-+-1e.0-[06]--+-0d.0
                        \-0d.1

00:1e.0 PCI bridge: Intel Corporation 82801 PCI Bridge (rev 90)
```