Platform Device Assignment to KVM-on-ARM Virtual Machines via VFIO

Antonios Motakis, Alvise Rigo, Daniel Raho
Virtual Open Systems
Grenoble, France
contact@virtualopensystems.com

***Abstract*—VFIO (Virtual Function I/O) is a Linux kernel infrastructure that allows to leverage the capabilities of modern IOMMUs to drive a device directly from userspace without any additional specialized kernel driver being involved. When used by QEMU/KVM, a device can be assigned to a guest VM, allowing to transparently handle all aspects of communication with the device, including DMA mapping, MMIO range mapping, and interrupts.**

To support a given hardware architecture and device, VFIO will need to be able to support the type of IOMMU that is in front of the device, and the discovery and configuration mechanisms of the bus that the device is connected to. However, often no auto configuration interface is exposed on ARM, as opposed to devices on a PCI bus, commonly used on x86.

In order to support VFIO on ARM, in this work platform devices support is being implemented as a VFIO_PLATFORM driver. Additionally, VFIO is being extended to support common IOMMUs found on ARM systems, such as the ARM SMMU. In this paper, we highlight the challenges met while implementing the required components, and extensions to VFIO and KVM in order to fully support device assignment to Virtual Machines on modern ARM systems.

***Keywords*-kvm; arm; virtualization; iommu; qemu; vfio; device passthrough; device assignment**

I. Introduction

The goal of this work is a functional implementation for device assignment to guest Operating Systems running under KVM on ARM platforms. Device assignment provides the most efficient way to do I/O, compared to other approaches such as device emulation, which imposes a high number of exits from guest context, and paravirtualization, which requires a specialized driver to be implemented in the guest.

The VFIO for ARM work described here targets the saveHSA architecture[1], which we will briefly introduce in this section. In Section II we introduce the KVM hypervisor used for this work, along with a brief description of its architecture and how its use of QEMU device emulation affects I/O performance. In Section III we will introduce the device assignment virtualization technique, which solves some of the problems of device emulation, and we discuss the main hardware requirement, the existence of an IOMMU, in Section III-B.

Having covered the targeted hardware environment, and the general considerations for device assignment, we will start covering the VFIO based implementation for KVM in Section IV. We will discuss our work on bringing VFIO on ARM platforms in Section V, covering the key implementation aspects that have been tackled compared to VFIO on x86. Finally, in Section VI we will be able to wrap up the work presented here, by discussing the implementation in QEMU of device assignment with VFIO for platform devices on ARM.

A. The saveHSA Architecture

The *SAVE Heterogeneous System Architecture* (saveHSA)[1] aims to introduce self adaptiveness and virtualization in heterogeneous system architectures. The target architectures (figure 1) feature heterogeneous computing resources in addition to CPUs, such as GPUs, *DFEs (Data Flow Engines)*, and other hardware accelerators. The project will introduce hardware virtualization to enable a concurrent resources allocation to different virtual machines using a self adaptive orchestrator. The main challenge for this kind of architecture is to enable Multiple VMs to share a computational core while keeping performance close to native.

This kind of architecture presents very demanding virtualization requirements, since hardware computational resources, such as GPUs, need to be able to be dynamically allocated by the hypervisor to different virtual machines. Multiple VMs might need to share a virtual computational core (e.g. vGPU), and moreover near native performance is expected by the system's user. Hardware accelerators, such as GPUs, when used to accelerate general purpose computing kernels present very high throughput and low latency requirements, which the hypervisor needs to comply to.

II. The Linux KVM Hypervisor

The *Linux Kernel Virtual Machine (KVM)[2]* is an established system virtualization solution, implemented as a driver running within Linux, which effectively turns the Linux kernel into a hypervisor. This approach takes advantage of the existing mechanisms within the Linux kernel, such as the scheduler, and memory management. This results in the KVM code base being very small compared to other hypervisors; this has allowed KVM to evolve at an impressive pace and become one of the most well regarded and feature full virtualization solutions.

KVM works by exposing a simple `ioctl` based interface to user space, through which a regular process can request to

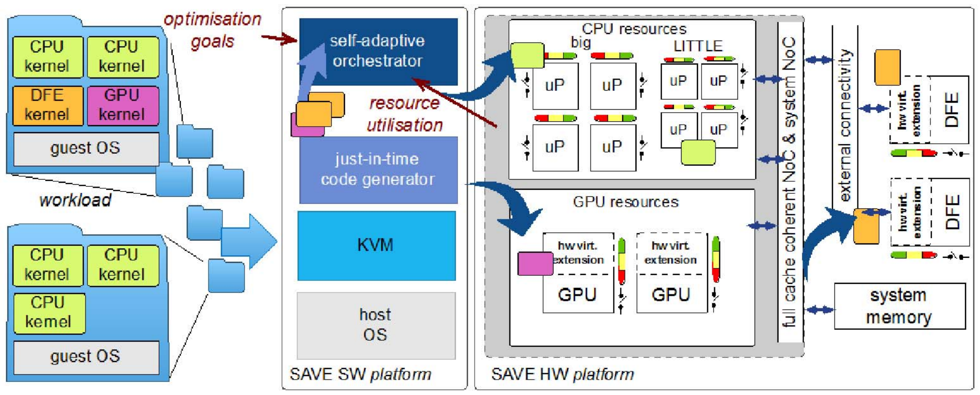

Figure 1. The Save Heterogeneous System Architecture (saveHSA)

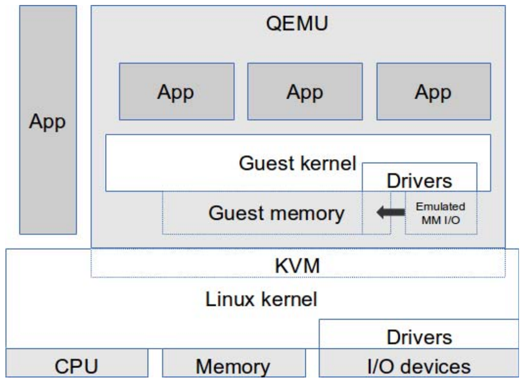

Figure 2. The KVM hypervisor architecture

be turned into a virtual machine. Usually *QEMU* is used on the user space side to emulate I/O devices, as seen in figure 2, with KVM handling virtual CPUs and memory management.

KVM itself handles the switching of the context of the processor when the process that corresponds to a virtual machine gets loaded by Linux, taking advantage of the virtualization extensions supported by the hardware. In this fashion the processor and the memory are virtualized, however to virtualize I/O devices, such as network interfaces and storage, the interface to user space is used, so these can be emulated by the application setting up the virtual machine (most commonly the QEMU emulator).

A. The User Space Driver: QEMU

QEMU[3] functions as a caller from user space for KVM, e.g. setting up the memory of the VM to be launched, the virtual CPUs to be used, et cetera. QEMU (using the interface exposed by KVM) configures memory regions that should trap when the guest attempts to read or write to them; the execution flow will return to QEMU, which will implement a number of I/O devices, such as network interfaces, graphics controllers, and storage, as well as user interface devices, such as keyboards and mice.

Depending on the underlying architecture QEMU may also handle injecting interrupts and emulating an interrupt controller in the same fashion; however since the interrupt controller is a critical component, on ARM systems KVM may take advantage of hardware virtualization support in the ARM Generic Interrupt Controller (ARM GIC).

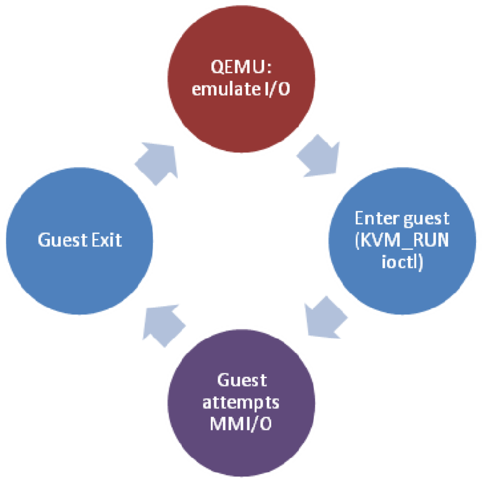

Figure 3. QEMU exits for MMIO emulation

B. Emulation of I/O Devices in QEMU

Even taking the exception of the ARM GIC into account, a typical ARM virtual machine still has a majority of its devices emulated by QEMU. This synergy between QEMU and KVM is based on the standard `ioctl` system call interface which KVM exposes to user space; QEMU simply issues `ioctl` commands to setup KVM, and to enter execution in guest context. In the case of a guest exit, QEMU is able to determine why the guest stopped executing and take appropriate action, e.g. by emulating a *Memory Mapped I/O (MMIO)* operation.

QEMU will then inject interrupts to the VM from the emulated device as needed, either by emulating the ARM GIC, or by taking advantage of the in kernel support for the ARM virtual GIC. Either way, execution will return to KVM, which will place the processor again in guest context.

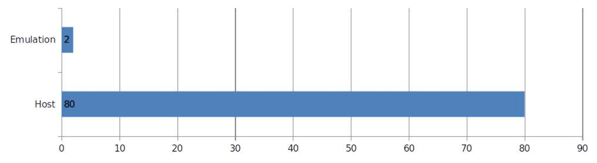

Figure 4. Cost of VM exits

QEMU emulated devices allow to use native drivers for the guest OS, as long as QEMU implements a device that the guest supports. However the frequent exits (figure 3) from guest execution context in order to emulate accesses to the device's MMIO regions have a significant performance cost, as seen in figure 4.[10]

C. Device Paravirtualization in QEMU

Another virtualization technique for I/O devices is called paravirtualization, which in QEMU is implemented using *Virtio*.[4] In this case, a full I/O device is not being emulated by QEMU; instead, a communication interface is implemented between QEMU and the guest OS. A specialized Virtio driver running in the guest, will directly coordinate with a backend running in QEMU. For example, a guest running a Virtio network driver may exchange network packets directly with the Virtio network backend running in QEMU.

This allows to avoid the high number of exits that are caused by emulating the interface of a real device; however, the guest OS needs to be modified to use a specialized Virtio driver. In contrast, with device assignment we will be able to use native drivers for a device, which is a significant advantage when targeting devices with proprietary binary drivers.

III. Introduction to Device Assignment

With device assignment, instead of emulating a device, or implementing a paravirtualization layer to exchange data with the guest, we can allow the guest to directly use an existing device present on the hardware, as seen in figure 5.[5] This allows the guest OS to use a native driver for that device; this is desirable especially in the case of GPU devices, where device drivers are often proprietary and shipped in binary form with those devices.

Having the guest OS directly access the device means significant performance improvement over the other approaches. Since exits from guest context to the hypervisor are not needed in order to emulate aspects of the device, a very low overhead for GPU use by Virtual Machines can be achieved. The expected result is very low latency and high throughput

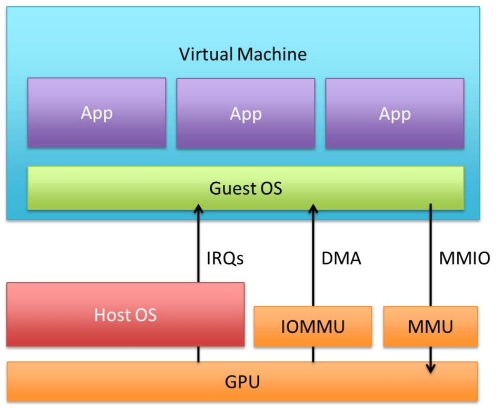

Figure 5. The device assignment approach lets the guest OS access a device directly

access from the guest driver to the physical device, with actual performance being very close to native.

Summing up, device assignment can realize both compatibility (native drivers) and performance requirements, since the guest accesses a given device directly, without interference from the hypervisor. However, as we will shortly see, this requires the presence of an IOMMU hardware device in front of the device we wish to assign to a virtual machine. The IOMMU is a necessary component, which allows to securely implement device assignment, without letting the target device interfere with the operation of the system outside the virtual machine using it.

A. DMA Operations of a Device Assigned to a Guest OS

Modern hardware accelerators include Direct Memory Access (DMA) capabilities, which allow them to quickly deliver data to the running operating system. Since hardware accelerators or GPUs do not reside behind an IOMMU, they operate with physical addresses preventing their seamless usage in Virtual Machines with kernel drivers.

DMA accesses are problematic when mapping a device either to a user space application or to a guest virtual machine.[5] Not only the hardware will be configured using virtual (or intermediate physical) memory addresses, other security concerns also apply since the guest operating system cannot be trusted with direct access to the host's or other VMs memory space.

Giving a guest OS direct control of a device which can perform DMA, will give a degree of freedom to the guest to set the addresses the device will perform DMA to. The device will be programmed for DMA using system *Physical Addresses,* which means that the guest being able to directly control the device, may corrupt the memory of the host system, or other virtual machines. Even worse, a malicious guest may take advantage of its capability to DMA to any memory in the system in order to purposely compromise the system.

The presence of an IOMMU on the system, allows to protect the system from accesses to memory from the device. The IOMMU will also translate the accesses from the device from the guest's *Intermediate Physical* to the *Physical* address space, allowing correct operation of the device.

B. The IOMMU Hardware

An IOMMU is a hardware block, comparable to the MMU of the CPU, which intercepts any memory access from programmable or not hardware accelerators, and translates the memory addresses, from virtual addresses to physical addresses. Hardware components behind a properly configured IOMMU will operate under the illusion that they are directly accessing the physical memory of the machine, while they have actually been mapped to the memory space of a virtual machine. Additionally, the IOMMU will prevent the associated devices from accessing memory beyond the mappings that have been assigned to them in the page tables.

Using IOMMUs in the platform, it is possible to extend the virtualization to the rest of the hardware enabling direct hardware resource assignment to a VM. In other words, hardware accelerators that are placed behind an IOMMU will benefit from seamless memory translation. A VM can now let a hardware accelerator to access the virtual address space of a user level process, or the intermediate physical address space of a VM.

C. Device Assignment with an IOMMU

The IOMMU is configured with a set of *page tables* which contain the mappings from virtual to physical addresses for each page of memory. Different implementations vary on details such as page sizes, page table format and whether there is a degree of compatibility with the CPU's page tables.

However it is not necessary for every hardware component in the platform to be implemented in a way where it is configured separately and isolated from other components. Several implementation dependent factors may affect isolation granularity. For example, components of an enclosure behind an IOMMU may perform transactions between them without having their traffic intercepted by the IOMMU. The IOMMU placement and the architecture of the system on-chip interconnect may also affect the IOMMU's operation.

In order to support systems with hardware components that may not be isolated from each other, the *IOMMU API* in the Linux kernel implements the concept of *IOMMU groups*. An IOMMU group is a set of devices which cannot be isolated from each other. This means that devices in the same group will always need to be assigned to the same virtual machine.

IV. Virtual Function I/O

VFIO (Virtual Function I/O) is a module taking advantage of these capabilities that has become part of the Linux kernel.[6], [7]

A user space program can take advantage of the VFIO API, and drive the device directly without a kernel driver being involved. The VFIO API allows mapping MMIO and DMA regions of memory to the user process securely, using a standard interface. It also allows the process to be notified when there is an incoming IRQ, completing the puzzle and allowing to fully drive the device from user space.

Of special interest for this work are the virtualization capabilities of VFIO, which allow virtual pass-through, i.e. the capability to directly assign devices to virtual machines, which can use an unmodified driver in order to use that device. The VFIO API provides a standard way for the hypervisor software to map any device to a guest (through QEMU in our case), allowing the guest to run an unmodified driver, while the host can handle and forward incoming interrupts. This is achieved thanks to the IOMMU's capability to translate the memory accesses of the device, from the intermediate physical address space shared between the device and the target VM, to real physical addresses.

In the x86 world and for PCI devices, KVM already support device assignment via VFIO;[9] we have performed the porting of this module, to ARM based architectures with platform bus devices.

A. IOMMU Groups and VFIO Containers

As stated before, in order to support systems with devices that perhaps cannot be isolated from each other, the IOMMU API in Linux implements the concept of groups. VFIO includes this concept, and groups are actually VFIO's ownership unit. However, VFIO extends the hierarchy by implementing the concept of *containers*; the user of the VFIO API may create a container and add one or more groups to it. This allows for multiple groups to share the same configuration; all devices assigned to the same VM will be placed under the same VFIO container.

B. Implementing Device Assignment with VFIO

In VFIO a bus specific module (the VFIO bus driver) binds to device resources, usually discoverable by some kind of bus specific mechanism, such as the PCI configuration space. Linux VFIO builds upon the existing IOMMU API, providing a standard interface where PCI devices behind an IOMMU can be unmapped from the host operating system, and subsequently mapped to another virtual address space.

When using the VFIO infrastructure, there are at least three modules involved that need to be loaded by the user.[7], [8] The first one is the core VFIO module, which implements most of the interface to user space and the management of VFIO groups and containers, while other functionality is abstracted into the other two modules. The VFIO core is architecture independent.

Two more key components that are specific to the target architecture:

- The first one is what VFIO refers to as the “bus driver”, which will bind to the device to be mapped, and register it with VFIO. Device resources that are to be remapped will be handled in this module. This includes mapping Memory Mapped I/O regions and forwarding device interrupts. For ARM based targets the a `VFIO_PLATFORM` driver is introduced as part of this work.
- The other component that needs tuning is the VFIO IOMMU driver, which will handle mapping DMA resources that require the IOMMU. These are resources that are handled with the granularity of a whole VFIO container (and the IOMMU groups it contains).

C. Binding a Device to VFIO

Before the user can set up VFIO to pass a device to a virtual machine or a user space driver, it needs to unbind all the devices that are under the same IOMMU group from the host kernel. Otherwise, since those devices need to share the same IOMMU configuration, unpredictable behavior may emerge.

Typically the IOMMU driver on the system, using the IOMMU API will expose this information under *SYSFS*. There are mainly two structures of interest to us: `/sys/bus/platform/devices/$device_id/iommu_group`, which is a symbolic link towards: `/sys/kernel/iommu_groups/$group_num`

The `iommu_group` directory will include links to all devices that are part of this group. In order to proceed, all those devices need to be unbound from their respective device drivers on the host.

Finally, for VFIO to manage a device, it should be bound to the VFIO bus driver, a module specific to the type of the device. The `VFIO_PLATFORM` module we introduce in Section V, allows the use of platform devices under VFIO. Devices can be bound to this module via `/sys/bus/platform/drivers/vfio-platform/bind`.

D. Base VFIO API

Initially, a container is created by opening the `/dev/vfio/vfio` character device. The file descriptor returned corresponds to that container, and can be managed by the opening process.

At the same time, under `/dev/vfio/` one character device per IOMMU group that VFIO knows about can be found. IOMMU groups are known to VFIO through the VFIO bus driver that is bound to each device to be assigned. Opening the corresponding character device also provides a file descriptor that can be used to manage this group with VFIO.

The VFIO API (figure 6) is partially device specific, as some calls are forwarded to the VFIO bus driver and the VFIO IOMMU driver. However, there are 3 key `ioctls` crucial to understanding how VFIO is used by user space applications. `VFIO_SET_CONTAINER` should be used on each group that needs to be used, in order to map it to the VFIO container that will handle it. IOMMU specific configuration will be then handled through the container file descriptor: the IOMMU driver must be specified with a `VFIO_SET_IOMMU ioctl`, which allows us to setup DMA memory that can be safely used thanks to the IOMMU, using a `VFIO_IOMMU_MAP_DMA` call. The interface for DMA mapping is specific to the VFIO IOMMU driver.

Typically memory regions and interrupt lines of a device can be mapped using `ioctl` functions on a *VFIO device file descriptor*; however since this is not directly accessible from the `/dev` file system, it needs to be requested using another specialized ioctl on the group file descriptor.

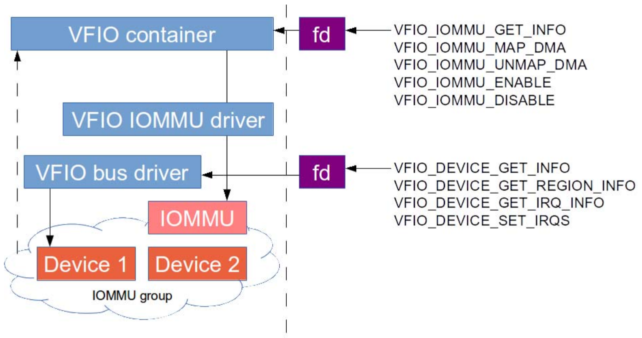

Figure 6. The main VFIO abstractions

This is where VFIO will invoke the hardware specific VFIO modules, the VFIO bus driver and VFIO IOMMU driver, which will implement device region mapping and DMA mapping `ioctl` calls respectively.

V. VFIO ON ARM FOR PLATFORM DEVICES

On ARM platforms PCI is often not used as the main interconnect, instead an on-chip interconnect or Network on Chip are generally used. The on-chip interconnects, typically do not have an auto configuration interface like PCI. Therefore, the available hardware components and their configurations, such as memory regions and allocated interrupts, need to be provided to the Linux kernel by another mechanism instead. One mechanism often used for this is a specially compiled *device tree* that described the target system, and is passed to the Linux kernel at boot time; alternatively, this information may be hardcoded inside the kernel for specific devices supported.

Overall, Linux treats devices like these as *platform devices*, instead of referring to a specific bus such as PCI. The key contribution of this work is the *VFIO platform* driver that has been developed, in addition to the implementation of the required extensions to make VFIO aware of ARM based IOMMUs, such as the ARM SMMU, using the `VFIO_IOMMU_TYPE1` driver.

In the reminder of this section, we will further describe the last two components mentioned and the changes performed to adapt those to the ARM architecture.

A. *VFIO Platform Bus Driver*

A VFIO bus specific driver needs to be bound to every device to be used with VFIO. There are several aspects of controlling a device that can vary widely between systems based on different architectures and device models. Therefore this specific driver needs to be aware of all of those aspects involved in driving the device, including what MMIO regions need to be accessible, and how interrupts are managed. The core VFIO module will delegate calls to memory map device regions, and handle interrupts to the bus specific VFIO driver. Additionally, most aspects specific to the device or the bus are implemented here.

The VFIO bus driver's functionality is implemented mostly on a device granularity, mainly with regard to memory mapping MMIO regions of the device, and implementing a way to forward interrupts.

When the user requests a VFIO device file descriptor, some functionality will be delegated to this module. The `read`, `write` and `mmap` functions are of particular interest, since they implement access to the MMIO regions of the device.

Several `ioctl` functions are implemented in this module, including getting the regions of the device. In the case of platform, the `ioctl` functions implemented directly are:

```
VFIO_DEVICE_GET_INFO
VFIO_DEVICE_GET_REGION_INFO
VFIO_DEVICE_GET_IRQ_INFO
VFIO_DEVICE_SET_IRQS
```

For example, the VFIO_DEVICE_GET_REGION_INFO call can provide details on the capabilities of each device region (as seen in `struct vfio_region_info` in algorithm 1). In addition, `VFIO_DEVICE_SET_IRQS` implements number of ways to interact with the IRQ signals of the device (see `struct vfio_irq_set` in algorithm 2). This allows to mask and unmask interrupts, and also to set an `eventfd` that the user can poll in order to implement an appropriate handler when a IRQ is being triggered.

It is important that this part of the API is implemented by the *bus driver*, since the nature of what regions describe is very specific to the device type. On different architectures it might describe MMIO regions, or PIO ports, and especially in the case of PCI that includes the PCI configuration space that is presenting additional implementation complexity. The VFIO platform driver likewise handles the MMIO regions and the interrupt types that are present on ARM devices.

B. *VFIO ARM IOMMU Support*

The other component of VFIO is the VFIO IOMMU specific module. Since there might be different IOMMU implementations that can be supported by VFIO in the future, this module implements calls to the IOMMU's functionality. Any `ioctl` to a container which is not recognized by VFIO will be forwarded to the VFIO IOMMU module, which can thus directly handle certain parts of the API, including for example `VFIO_CHECK_EXTENSION` and `VFIO_IOMMU_MAP_DMA`, which handles the mapping of regions to the IOMMU.

VFIO implements the `VFIO_TYPE1_IOMMU` driver type, which describes a generic IOMMU, implementing the IOMMU API under the assumption that it does not have restrictions on the virtual address ranges that can be mapped to. The user can set this driver type for a container using the `VFIO_SET_IOMMU ioctl` call.

The functionality actually implemented by the `VFIO_TYPE1_IOMMU` driver type, mainly amounts to 4 `ioctl` functions, directly passed to it:

```
VFIO_CHECK_EXTENSION
VFIO_IOMMU_GET_INFO
VFIO_IOMMU_MAP_DMA
VFIO_IOMMU_UNMAP_DMA
```

All in all, compared to the bus driver, the VFIO IOMMU driver has a simpler interface to implement. However, there are still many internal considerations that apply when mapping a DMA memory region to the IOMMU. In particular, the `VFIO_TYPE1_IOMMU` driver in `vfio_dma_map` will break the requested memory range into individual pages, and pin each one to memory, so they can't be swapped to the disk while the user space process using VFIO might perform a DMA operation.

`VFIO_TYPE1_IOMMU` was intended to be an interface to any IOMMU implementing the Linux IOMMU API, as architecture independent as possible. The *ARM SMMU*s used while implementing `VFIO_PLATFORM` are implementing this API, so they are in principle compatible with `VFIO_TYPE1_IOMMU`, with a few adjustments. The ARM SMMU driver was patched to expose IOMMU groups via the IOMMU API. Several bugs in that driver were discovered and fixed while integrating `VFIO_TYPE1_IOMMU` on a system using an ARM SMMU.

VI. Implementing Device Assignment for QEMU/KVM with VFIO

Having all the above described functionality exposed by the kernel via the VFIO API, implementing a QEMU device based on VFIO can complete the puzzle, as in figure 7. For this purpose, a *PL330 DMA Controller device for QEMU* has been implemented, which is based on VFIO and demonstrates full device assignment to a KVM guest of a *ARM PL330 DMA Controller*.

A. *QEMU PL330 VFIO Based Device Prototype*

The implemented components that validate the development efforts for device assignment on ARM include: A DMA driver for the PL330 based entirely in user space, realized thanks to the VFIO API and our implemented

Algorithm 1 struct vfio_region_info

```
struct vfio_region_info {
  __u32 argsz; __u32 flags;
  #define VFIO_REGION_INFO_FLAG_READ  (1 << 0)
  #define VFIO_REGION_INFO_FLAG_WRITE (1 << 1)
  #define VFIO_REGION_INFO_FLAG_MMAP  (1 << 2)
  __u32 index;        /* Region index */
  __u32 resv;         /* Reserved for alignment */
  __u64 size;         /* Region size (bytes) */
  __u64 offset;       /* Region offset from start of device fd */
};
```

Algorithm 2 struct vfio_irq_set

```
struct vfio_irq_set {
   __u32 argsz;
   __u32 flags;
   #define VFIO_IRQ_SET_DATA_NONE        (1 << 0) /* Data not present */
   #define VFIO_IRQ_SET_DATA_BOOL        (1 << 1) /* Data is bool (u8) */
   #define VFIO_IRQ_SET_DATA_EVENTFD     (1 << 2) /* Data is eventfd (s32) */
   #define VFIO_IRQ_SET_ACTION_MASK      (1 << 3) /* Mask interrupt */
   #define VFIO_IRQ_SET_ACTION_UNMASK    (1 << 4) /* Unmask interrupt */
   #define VFIO_IRQ_SET_ACTION_TRIGGER   (1 << 5) /* Trigger interrupt */
   __u32 index;
   __u32 start;
   __u32 count;
   __u8 data[];
};
```

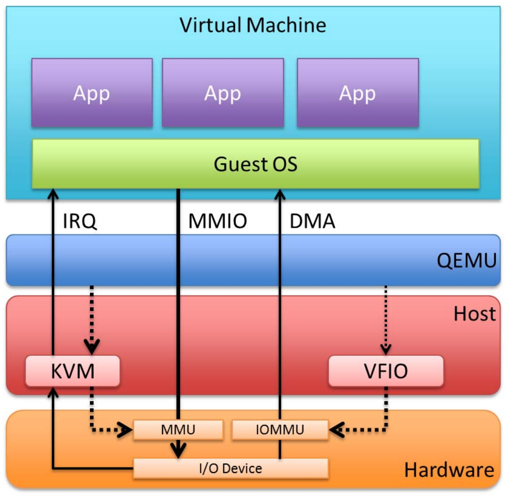

Figure 7. Device assignment with VFIO and QEMU/KVM

VFIO_PLATFORM driver. This driver stresses the functionality implemented in VFIO_PLATFORM, including handling interrupts, DMA mapping and accessing the DMA controller's MMIO registers.

The QEMU PL330 VFIO based device, which exposes one PL330 DMA Controller to the guest. This QEMU device does not emulate a DMA controller; it maps one of the host's hardware DMA controllers directly to the guest, using the API we have described previously.

A Linux kernel DMA test for the PL330 DMA Controller, validates that the devices has been assigned to the guest successfully, and operates correctly. This demonstrates that the guest operating system is accessing the DMA Controller directly, without being aware of the intervention of VFIO and the host hypervisor. The guest operating system and DMA driver are completely unmodified.

VII. Conclusion

Overall, this work demonstrates successful device assignment on an ARM based platform and with KVM for the first time. The ARM PL330 DMA controller can be mapped directly to a guest, validating the case for VFIO on ARM. In addition to virtualization, VFIO enables native performance in user space drivers which can be used in HPC [11] and embedded systems.

In the future, target devices may implement multiple virtual functions to be shared between multiple virtual machines efficiently, making VFIO a very lucrative solution for device virtualization.

Also, future IOMMUs on ARM systems, will support two stages of memory translation, with only the second stage used by VFIO. This will allow the guest OS to also take advantage of the IOMMU seamlessly.

Our VFIO work on ARM, featuring the `VFIO_PLATFORM` module is an ongoing work and is aiming to be included into future upstream versions of the Linux kernel. Follow up activities may examine the performance characteristics of VFIO with future devices on ARM which will include an SMMU.

Acknowledgement

This work has been conducted under the SAVE FP7 project[1], and is based on a number of open source technologies discussed here.

References

[1] G. Durelli, M. Coppola, K. Djafarian, G. Kornaros, A. Miele, M. Paolino, O. Pell, C. Plessl, M.D. Santambrogio, C. Bolchini, *SAVE: Towards efficient resource management in heterogeneous system architectures*, in Proc. Int. Symp. Applied Reconfigurable Computing, ARC, pp. 1-7, 2014

[2] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, *kvm: the Linux virtual machine monitor*, in Proceedings of the Linux Symposium (Vol. 1, pp. 225-230), 2007

[3] F. Bellard, *Qemu, a Fast and Portable Dynamic Translator*, Usenix annual technical conference, 2005

[4] R. Russell, *virtio: towards a de-facto standard for virtual I/O devices*, ACM SIGOPS Operating Systems Review 42.5 (2008): 95-103.

[5] B. Yassour, M. Ben-Yehuda, O. Wasserman, *Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines*, IBM Research Division Haifa Research Laboratory, September 20, 2008

[6] A. Williamson, *VFIO core framework*, http://lwn.net/Articles/473975/

[7] J. Corbet, *Safe device assignment with VFIO* http://lwn.net/Articles/474088/

[8] A. Williamson, *VFIO - "Virtual Function I/O" https://www.kernel.org/doc/Documentation/vfio.txt*

[9] A. Williamson, *VFIO: Are we there yet?* Linux Plumbers Conference 2012, August 29-31 2012

[10] A. Motakis, A. Spyridakis, D. Raho, *Introduction on performance analysis and profiling methodologies for KVM on ARM virtualization*, Proc. SPIE 8764, VLSI Circuits and Systems VI, 87640N (May 28, 2013)

[11] J. Squyres, *SC'11 Cisco booth demo: Open MPI over Linux VFIO*, http://blogs.cisco.com/performance/open-mpi-over-linux-vfio/