Hardware-Assisted Virtual IOMMU with Nested Translation

Siqi Zhao[1]

[1]T-Head Semiconductor Co., Ltd.

Abstract

Passthrough devices is the default option for high-performance I/O for virtual machines. However, a virtual machine cannot efficiently nor transparently take advantage of a virtual IOMMU to manage device assigned to it, due to the fact that existing solutions for virtual IOMMU is based on emulation or paravirtualization. This extended abstract describes a hardware-assisted design for providing efficient and transparent virtual IOMMU for virtual machines.

Introduction

IOMMU is a fundamental infrastructural component used in virtualization application scenarios such as cloud servers, edge servers or home servers. In such scenarios, the I/O performance of virtual machines cannot match that of the bare-metal machines since I/O devices are typically emulated or paravirtualizaed. To improve I/O performance, physical devices can be directly assigned to virtual machines with assistance of an IOMMU; a technique commonly known as 'device passthrough'. The passthrough devices can be directly accessed by the guest kernel with Guest Physical Address (GPA). However, existing IOMMU infrastructure provides limited support for virtual IOMMU. The virtual machine cannot efficiently take advantage of the benefits of an IOMMU, albeit being virtual, such as non-contiguous I/O buffer, restricted memory range accessible by the device or user-mode direct access inside the virtual machine.

The existing IOMMU infrastructure relies on two approaches to provide virtual IOMMU support to virtual machines. The first approach is emulation and the second approach is paravirtualization.

In the emulation approach, the host relies on the traps generated by guest's access to the MMIO range occupied by the virtual IOMMU to trigger the execution of the emulation logic (typically within QEMU) for the virtual IOMMU. The emulation logic maintains the necessary set of hardware state expected by the guest's IOMMU driver. As an important step, the emulation logic needs to track the modification of the mappings performed by the guest's driver in order to combine the mappings with the second stage address translation mappings configured by the host.

The reason for this tracking and combination is that there is only one stage of translation in the hardware IOMMU. Specifically, the emulation logic needs to obtain the GPA for which a guest virtual address (GVA) is mapped to by walking the guest's I/O translation tables, then obtain the host physical address (HPA) for which the GPA is mapped to by walking the host's second stage translation table, before finally fill the mapping from the GVA to HPA into the one stage of translation tables used by hardware. The emulation process is involved and lengthy. Every time the guest modifies the mapping, the host undergoes this process. Figure 1 illustrates this architecture. Certain architecture supports nested translation, however, the first stage of translation tables are still configured by the host, the trapping and emulation is unchanged.

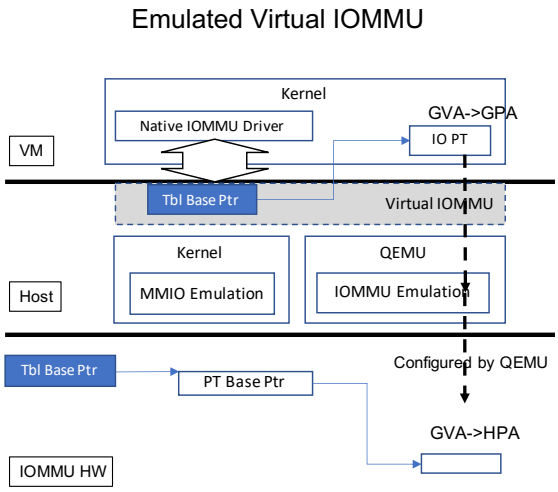

Figure 1: *Emulated Virtual IOMMU*

In the paravirtualization approach, i.e. `virtio` based virtual IOMMU, the hardware programming model of the IOMMU is replaced by a simple software-defined interface. Instead of trapping, the guest explicitly communicates the mappings that it requires to the host via the interface. The host then walks the second stage of page tables to obtain the GPA to HPA mappings, before finally configure the combined mapping in the translation tables used by hardware. This approach does not necessarily involve the trapping and emulation, however, it requires explicit support

from both the guest and host. Figure 2 illustrates this architecture.

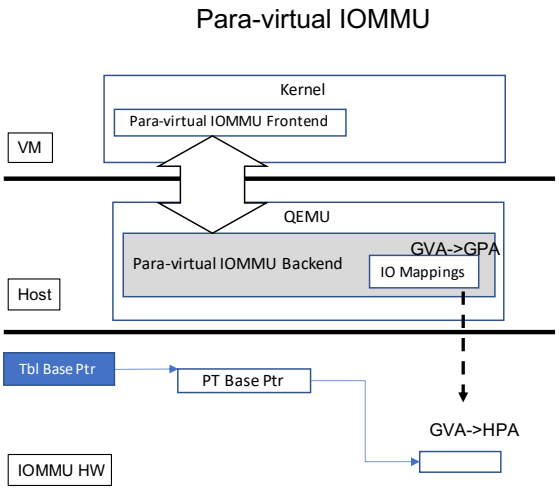

Figure 2: *Para-virtual IOMMU*

The Design of Hardware-Assisted Virtual IOMMU

In this work, we propose an IOMMU architecture that is capable of directly exposing the hardware IOMMU to the virtual machines. The guest is presented with an IOMMU that is identical to the physical IOMMU, therefore, it can directly reuse the driver written for the host. Furthermore the guest's translation tables are directly used by the hardware IOMMU to translate DMA addresses, eliminating any need to synchronize and combine translation mappings.

The design uses a memory-resident region to store the register state of the virtual IOMMU. The data at a given offset within the region represents the value of the register of the virtual IOMMU at that offset within the hardware MMIO range. This page is mapped to the guest in the G-stage mappings, therefore, the guest's IOMMU driver can directly access the page. For example, the guest fills the GPA of the root of its translation tables into this page when the driver configures the root register. Figure 3 illustrates this architecture.

When the hardware IOMMU is configured to perform nested translation for a given device, the device's translation tables store a pointer to the aforementioned virtual IOMMU stage region along side the pointer to the G-stage page tables. After obtaining the pointer to the memory region when the hardware walks the translation tables, it follows the translation tables configured by the guest via the root pointer filled by the guest, treating all the addresses as GPA and translating them via walking the G-stage page tables. In the end, the GVA in the DMA address is translated to the

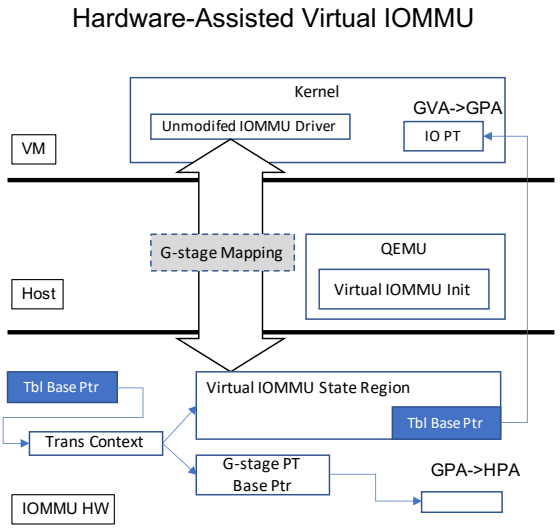

Figure 3: *Hardware-Assisted Virtual IOMMU*

HPA, and the transaction is forwarded to the fabric for memory access.

Implementation

We have implemented our design in QEMU and Linux KVM. We have also completed necessary modification in the Linux kernel. The QEMU involves the emulation logic for our IOMMU as well as necessary modification on the VFIO related configurations when QEMU is used by the KVM. We implemented our IOMMU driver in the Linux kernel, and performed minor modification on the VFIO kernel framework. The same driver works for both the host and the guest.

Discussion

The design presented in this work is transparent to software. The complex trapping and emulation is eliminated. It allows transparent virtualization of the IOMMU, simplifying porting and adoption efforts.

The memory region of virtual IOMMU state allows support for large number of virtual machines, compared to register-based approach. There is no need for the host the switch register contents according to the scheduling of the host.

Our design is not coupled with any specific bus features such as the Address Translation Service of PCIe, potentially allowing for wider range of adoption.