Lenovo

An Introduction to IOMMU Infrastructure in the Linux Kernel

Introduces the use of IOMMU in Linux to improve performance

Describes two IOMMU modes (DMA translation mode and pass-through mode) in Linux

Shows default IOMMU mode in Linux OSes

Provides the step-by-step instruction about how to configure a direct device access for a guest OS

Adrian Huang

LENOVO PRESS

Abstract

The Input-Output Memory Management Unit (IOMMU) is a component in a memory controller that translates device virtual addresses (can be also called I/O addresses or device addresses) to physical addresses. The concept of IOMMU is similar to Memory Management Unit (MMU). The difference between IOMMU and MMU is that IOMMU translates device virtual addresses to physical addresses while MMU translates CPU virtual addresses to physical addresses.

This paper explains the IOMMU technology, providing a high-level overview of IOMMU and IOMMU infrastructure in Linux kernel. Two IOMMU kernel modes (DMA translation mode and pass-through mode) are then described in detail. The last section of the white paper illustrates IOMMU use case with the PCI pass-through device in virtualization environment.

This paper is intended for IT specialists who want to know the difference between IOMMU DMA translation mode and IOMMU pass-through mode by means of the high-level overview, and should have knowledge of how to configure the Linux kernel and a familiarity with virtualization technologies such as KVM and Xen. The paper is also suitable for software developers who want to know the Linux kernel IOMMU subsystem, and it is recommended that they already have kernel development experience and knowledge of how MMU works.

Contents

Introduction

In a virtualization environment, the I/O operations of I/O devices of a guest OS are translated by they hypervisor (software-based I/O address translation). This behavior results in a negative performance impact. The Input-Output Memory Management Unit (IOMMU) is a hardware component that performs address translation from I/O device virtual addresses to physical addresses. This hardware-assisted I/O address translation dramatically improves the system performance within a virtual environment.

This paper covers the following items:

- PCI device: two PCI device virtualization models in a virtualization environment: emulation model and pass-through model
- IOMMU subsystem in Linux kernel
- Difference between IOMMU DMA translation mode and IOMMU pass-through mode
- Using a lab configuration to show the use of IOMMU with a direct access device in a guest OS: The I/O operations will be translated by IOMMU.

PCI Device Virtualization Models

The two PCI Device Virtualization models are Emulation model and Pass-through model.

Emulation Model (Hypervisor-based device emulation)

Figure 1 illustrates the emulation model of the PCI device virtualization. The hypervisor needs to manipulate the interaction between the guest OS and the associated physical device. It implies that the hypervisor translates device address (from device-visible virtual address to device-visible physical, and vice versa), which requires more CPU computation power and impacts the system performance when heavy I/O occurs.

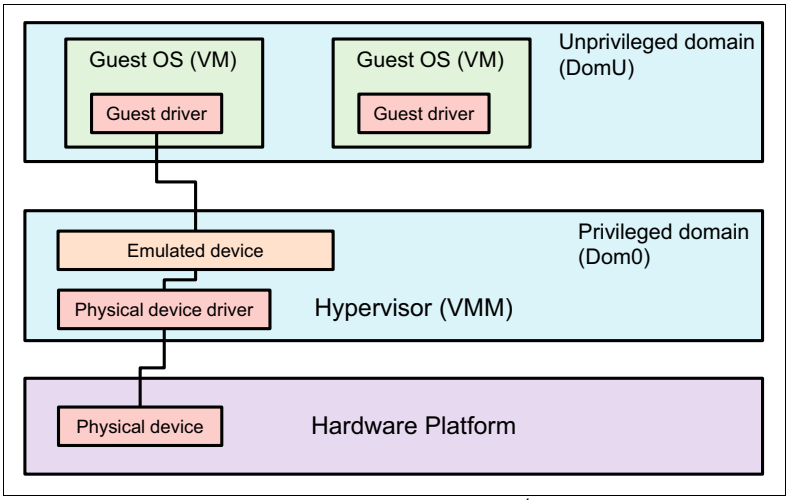

Figure 1 Device Virtualization Model: Emulation[1]

[1] From https://developer.ibm.com/tutorials/l-pci-passthrough/

Pass-through Model

The right-hand side of Figure 2 illustrates the model that the hypervisor is bypassed for the interaction between the guest OS and the physical device. The hypervisor does not need to deploy the dedicated software for emulating the physical device and translating device address. This model improves the system performance by means of a hardware-assisted component.

Intel names the hardware-assisted component “Intel Virtualization Technology for Directed I/O (VT-d)”, whereas AMD titles it “AMD I/O Memory Management Unit (IOMMU) or AMD I/O Virtualization Technology (AMD-Vi)”.

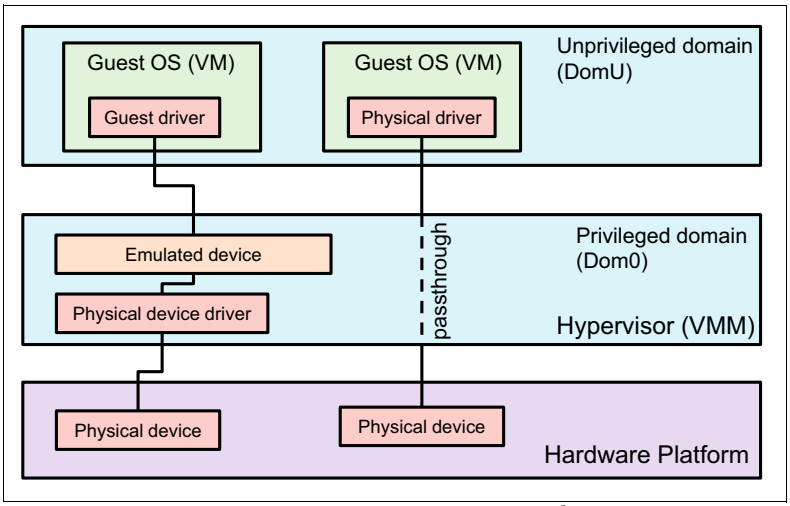

Figure 2 Device Virtualization Model: Pass-through[2]

[2] From https://developer.ibm.com/tutorials/l-pci-passthrough/

MMU and IOMMU

The aim of MMU (Memory Management Unit) is to translate CPU-visible virtual address to physical address. The purpose of IOMMU is similar to that of MMU. The translated virtual address of IOMMU is device-visible virtual address instead of the CPU-visible one. Figure 3 illustrates PCI pass-through model by leveraging the IOMMU hardware.

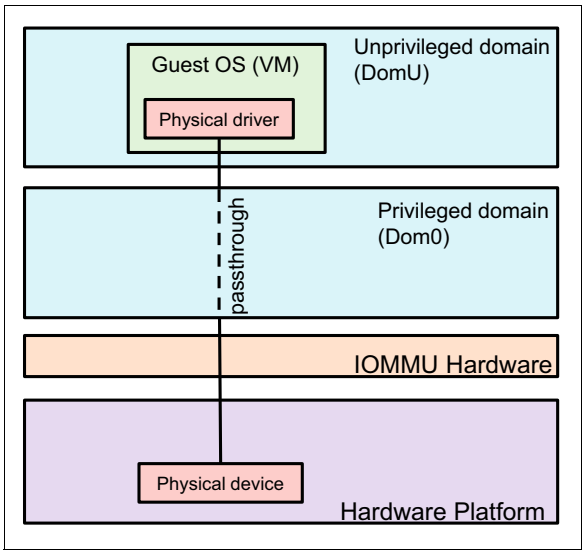

Figure 3 PCI Passthrough Device Example via IOMMU Hardware

The IOMMU hardware includes two functionalities:

- DMA remapping functionality manipulates address translation for PCI devices
- Interrupt remapping functionality routes interrupts of PCI devices to the corresponding guest OSes.

This paper focuses on DMA remapping functionality.

IOMMU Subsystem in Linux Kernel

This section describes the high-level overview of the IOMMU subsystem in Linux kernel and illustrates how I/O requests are propagated in the Linux kernel. In order to understand the I/O address translation procedure, the I/O page table and the data structure are illustrated.

The operating system needs to understand the IOMMU hardware information, so the system firmware provides the IOMMU description by means of an ACPI table. This will be also discussed in this section.

IOMMU Subsystem in Linux Kernel – High-level Overview

Figure 4 illustrates the high-level overview about IOMMU subsystem in Linux kernel.

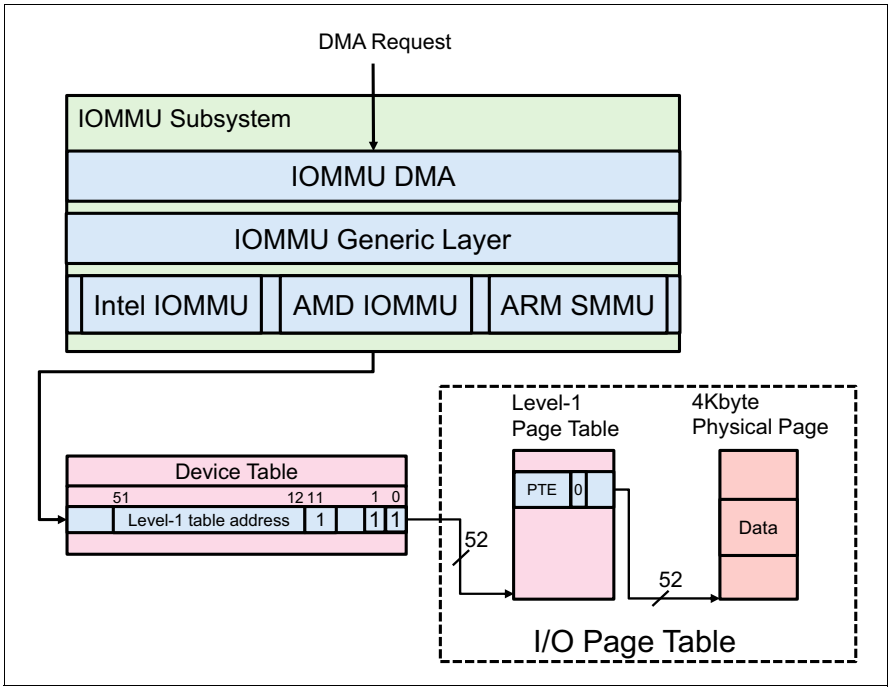

Figure 4 IOMMU Subsystem in Linux Kernel: High-level Overview

The subsystem contains three layers:

- IOMMU DMA Layer: This layer receives the DMA requests from I/O devices and forwards the request to IOMMU generic layer. It is the glue layer between DMA-API and IOMMU-API.
- IOMMU Generic Layer (or IOMMU-API Layer): This layer provides generic IOMMU APIs for interaction with IOMMU DMA layer and hardware specific IOMMU layer.
- Hardware Specific IOMMU layer: This is a hardware-specific driver in order to interact with the underlying IOMMU hardware. It also configures the proper I/O page table based on the requested DMA address so that IOMMU hardware can translate DMA address correctly.

  IOMMU hardware reports the exception event if the requested DMA address cannot be translated successfully. Some fields are configured either 0 or 1 in Figure 4. Refer to section 2.2.6 “I/O Page Tables for Guest Translations” in AMD I/O Virtualization Technology (IOMMU) Specification, available from:

  https://www.amd.com/system/files/TechDocs/48882_IOMMU_3.05_PUB.pdf

When will IOMMU hardware-specific layer update the I/O Page Table? Two cases are available:

- Direct mapping (or identity mapping) defined in Advanced Configuration and Power Interface (ACPI) table.

  When probing/initializing IOMMU hardware, IOMMU hardware-specific layer parses the direct mapping information stored in ACPI table and configures I/O page table based on ACPI table.

- DMA requests from I/O devices.

  When DMA requests are initiated from I/O devices, Linux kernel forwards and processes them in device driver, DMA subsystem and IOMMU subsystem as shown in Figure 5.

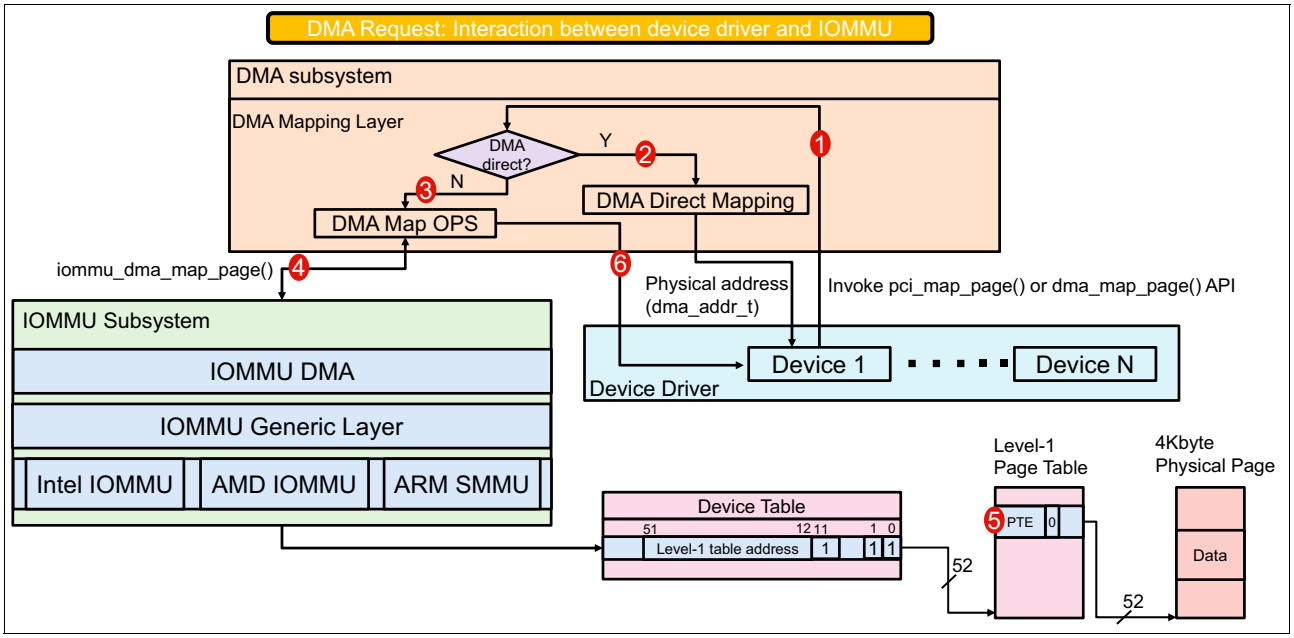

Figure 5 DMA Request: Interaction between Device Driver, DMA Subsystem and IOMMU Subsystem

The process steps 1-5 in Figure 5 are as follows:

1. Device driver invokes pci_map_page() or dma_map_page() API to get the physical address. Note that the physical address means the guest physical address if the running OS is in virtualization environment.
2. If the DMA request is directly mapped, DMA subsystem returns the calculated physical address to device driver directly.
3. If the DMA request is not directly mapped, the DMA request is forwarded to IOMMU subsystem.
4. DMA subsystem invokes iommu_dma_map_page() to request IOMMU subsystem to map virtual address to physical address.
5. IOMMU subsystem maps virtual address to physical address and configures the corresponding I/O page table so that IOMMU hardware can proceed address translation correctly.

Two-level Address Translation

IOMMU hardware supports two-level address translation:

- Guest translation, which maps Guest Virtual Address (GVA) to Guest Physical Address (GPA)
- Host translation (or nested translation), which maps GPA to System Physical Address (SPA).

Figure 6 illustrates 4K-byte page translation with 4-level I/O page table. This translates GVA to GPA. The addresses of the GCR3 table and the level-4 table address are SPAs, and those of PM4E, PDPE (Page Directory Pointer Entry), PDE (Page Directory Entry) and PTE (Page Table Entry) are GPAs. This implies that those GPAs needs to be translated to SPA in order to get the page table data from physical memory. Nested address translation (or host translation) achieves the requirement. Linux IOMMU subsystem constructs I/O page table and GCR3 table so that IOMMU hardware can deal with DMA translation properly.

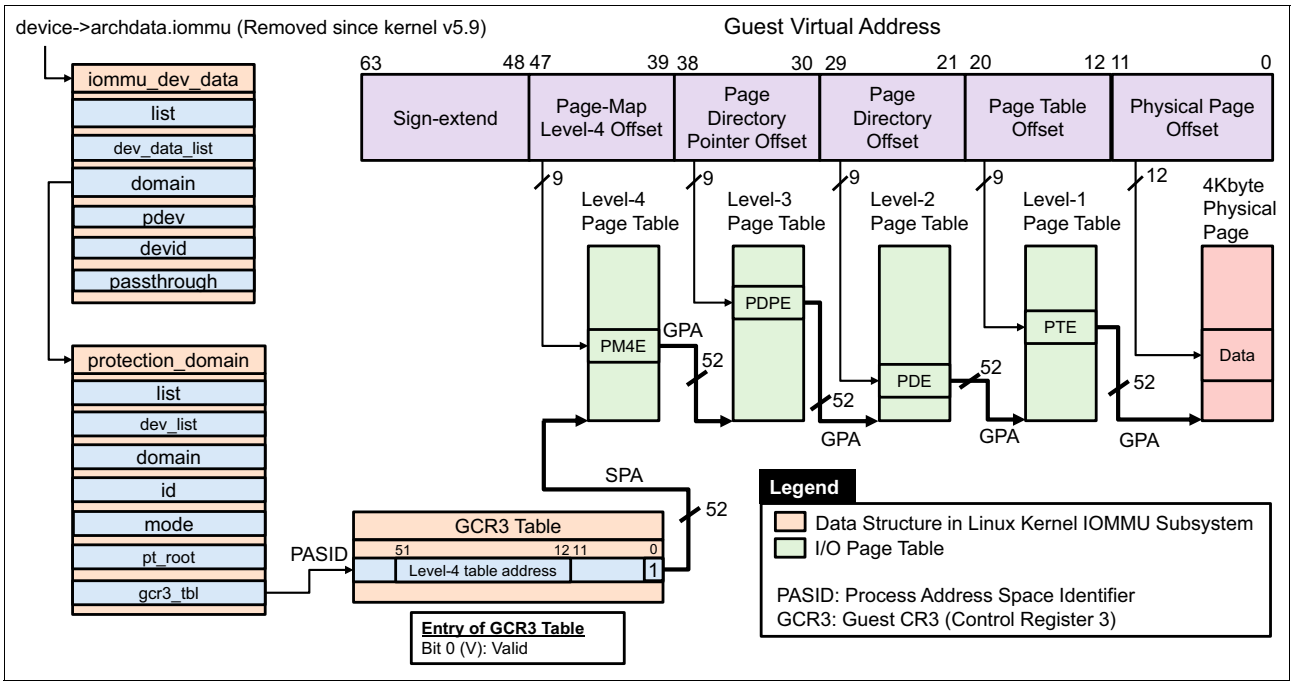

Figure 6 GVA to GPA Translation: 4K-byte Page Translation with 4-level I/O page table

Figure 7 illustrates GPA to SPA translation with 4K-byte page translation. The example shows 3-level page table translation. The addresses of level-3 table address and all PDEs are SPAs. Either a PDE indicating to the next-level page table or a PTE pointing to the system physical memory address is configured in each page table entry. Bit 9-11 (next level field) of a PDE or PTE indicates the next level of the page table. The next level field of a PTE is either 0 or 7, which indicates the end of the address translation process.

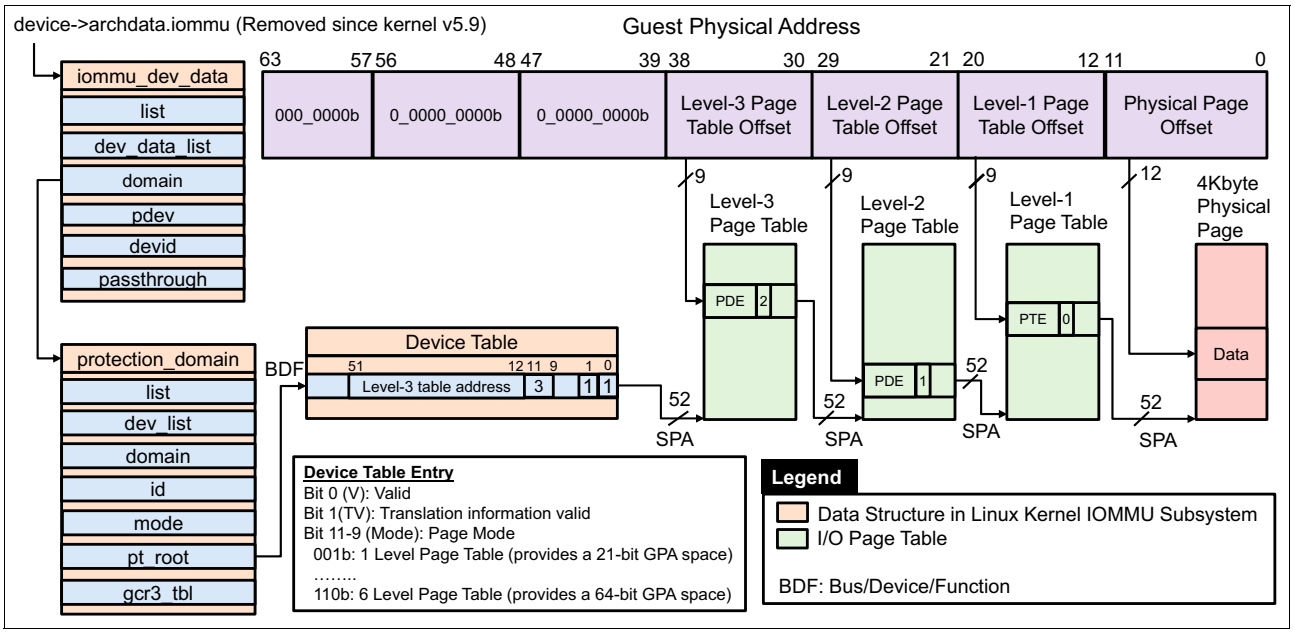

Figure 7 GPA to SPA translation: 4K-byte Page Translation with 3-level I/O page table

IOMMU Description in ACPI

The system platform firmware defines a data structure describing I/O Virtualization. This structure is named DMA Remapping Reporting (DMAR) by Intel and I/O Virtualization Reporting Structure (IVRS) by AMD. It resides in ACPI table that is used to notify OS software about IOMMU capabilities and configuration in the platform.

Refer to the AMD and Intel specification for details:

- AMD I/O Virtualization Technology (IOMMU) Specification

  https://www.amd.com/system/files/TechDocs/48882_IOMMU_3.05_PUB.pdf

- Intel Virtualization Technology for Directed I/O Architecture Specification

  https://software.intel.com/content/www/us/en/develop/download/intel-virtualization-technology-for-directed-io-architecture-specification.html

When initializing IOMMU hardware in Linux IOMMU subsystem, the IOMMU driver parses IVRS from ACPI table. If IVRS does not exist in the system, the IOMMU driver ignores the initialization flow. Conversely, the IOMMU driver initializes IOMMU hardware based on IVRS that includes one or more I/O Virtualization Definition Blocks (IVDBs).

Two types of IVDBs are as follows:

- I/O Virtualization Hardware Definition (IVHD): An IVHD describes the capabilities and configuration of IOMMU hardware as well as system I/O topology associated with each IOMMU hardware.
- I/O Virtualization Memory Definition (IVMD): An IVMD describes the special memory constraints for specific devices.

Figure 8 illustrates AMD IOMMU hardware description known as IVHD. The figure shows two IVHDs in the system, and the corresponding devices are attached to each IVHD. The detail of IVHD is elaborately described in AMD IOMMU specification.

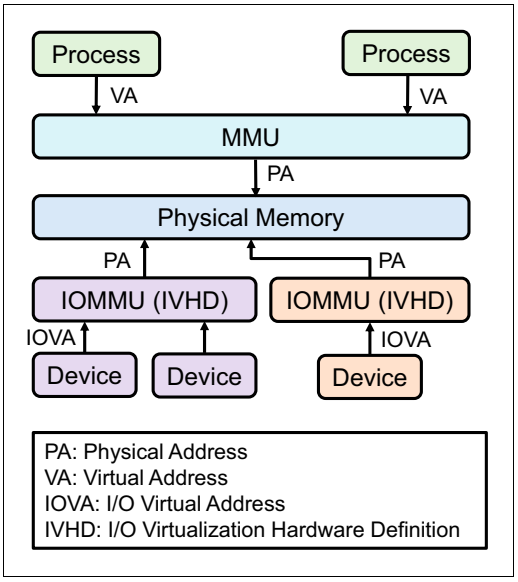

Figure 8 AMD IOMMU Hardware Description – IVHD

Linux Kernel IOMMU: DMA Translation Mode versus Pass-through Mode

This section describes the differences between IOMMU DMA translation mode and IOMMU pass-through mode in Linux kernel. Figure 9 shows DMA translation mode on the left and Pass-through mode on the right.

IOMMU pass-through mode is widely enabled in virtualization environment. This section also lists the default IOMMU operation mode of Linux OSes and provides a kernel parameter to change the IOMMU mode.

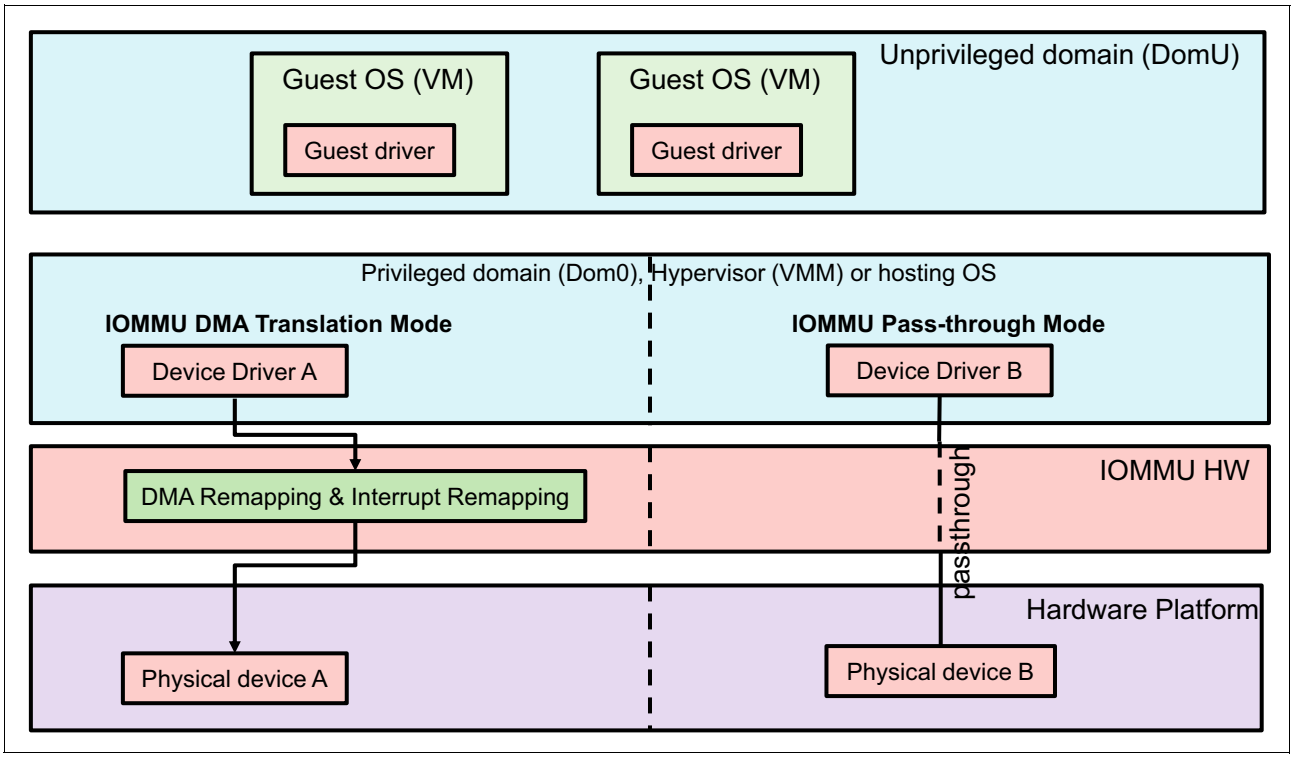

Figure 9 DMA Translation Mode and Pass-through Mode

The modes are as follows:

- IOMMU DMA Translation Mode

  Figure 9 on the left side shows the IOMMU DMA translation mode. It means that the hosting OS (hypervisor) applies IOMMU-backed operations for DMA translation. In other words, the IOMMU driver of the hosting OS examines all DMA requests and configures the corresponding IOMMU hardware so that IOMMU hardware can translate those requests correctly.

- IOMMU Pass-through Mode

  Figure 9 on the right side illustrates the IOMMU Pass-through mode. It bypasses the DMA translation from the hypervisor's point of view, which means DMA addresses equals to system physical addresses. This mode along with enabling PCI pass-through model (see "Pass-through Model" on page 4) is widely adopted in virtualization environment as shown in Figure 10 on page 12. The following FAQs explain concerns about IOMMU Pass-through mode.

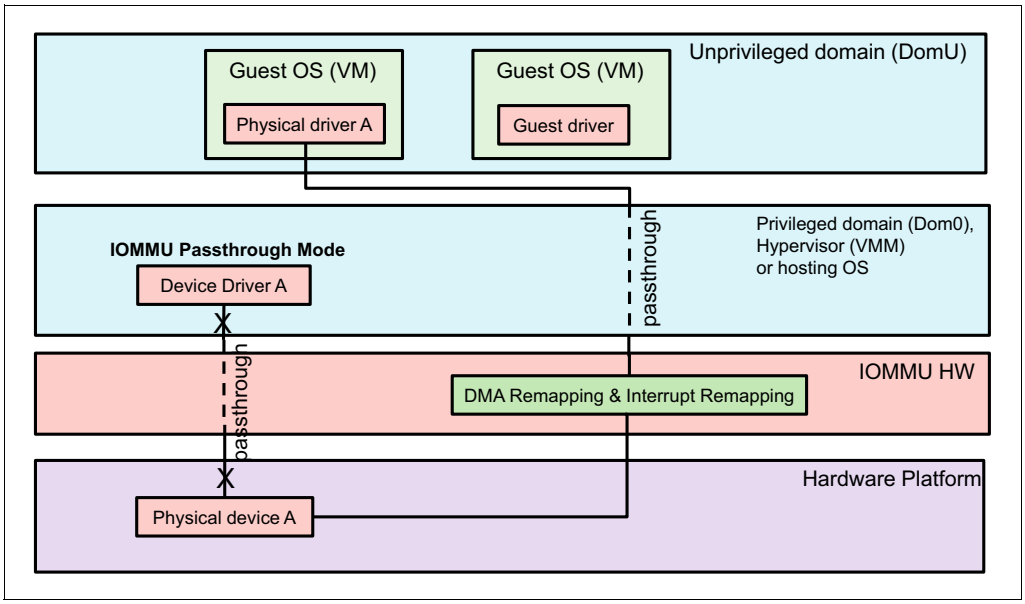

Figure 10 IOMMU Pass-through Mode in Virtualization Environment – Direct Device Access

What is the difference between PCI pass-through and IOMMU pass-through?

PCI pass-through model bypasses the hypervisor's intervention to render the guest OS to take control of the physical device directly. IOMMU pass-through mode bypasses the DMA translation from the hypervisor. The hypervisor does not need to process DMA requests when IOMMU pass-through mode is enabled in Linux. PCI pass-through and IOMMU pass-through work collaboratively to enable the guest OS to have the direct control of the physical device.

What is the difference between IOMMU pass-through mode and disabling IOMMU option in BIOS setup?

Disabling IOMMU option in BIOS setup means the IOMMU hardware is not exported to OS software because the IOMMU related data structures are not embedded in the ACPI table. Therefore, OS software cannot interact with IOMMU hardware. In this circumstance, the DMA address equals to the system physical address (no DMA translation is required) in the hypervisor.

IOMMU pass-through mode and disabling IOMMU have the same symptom – the DMA address equals to the system physical address. The main difference is that the guest OS can have the direct device access with the aid of the IOMMU pass-through mode, whereas the guest OS cannot have the direct device access when disabling IOMMU option in BIOS setup.

Thanks to the IOMMU pass-through mode and the PCI pass-through model, the guest OS can directly access the physical device without any SW changes. Apparently, the hosting OS requires a specific component interacting between the guest OS and the physical device.

Virtual Function I/O (VFIO) framework running on the hypervisor aims at providing user-space application for direct device access. QEMU, a user-space application, leverages VFIO framework to expose the direct access of the physical device to the guest OS. Figure 11 illustrates how VFIO framework cooperates with the guest OS, PCI driver and IOMMU driver.

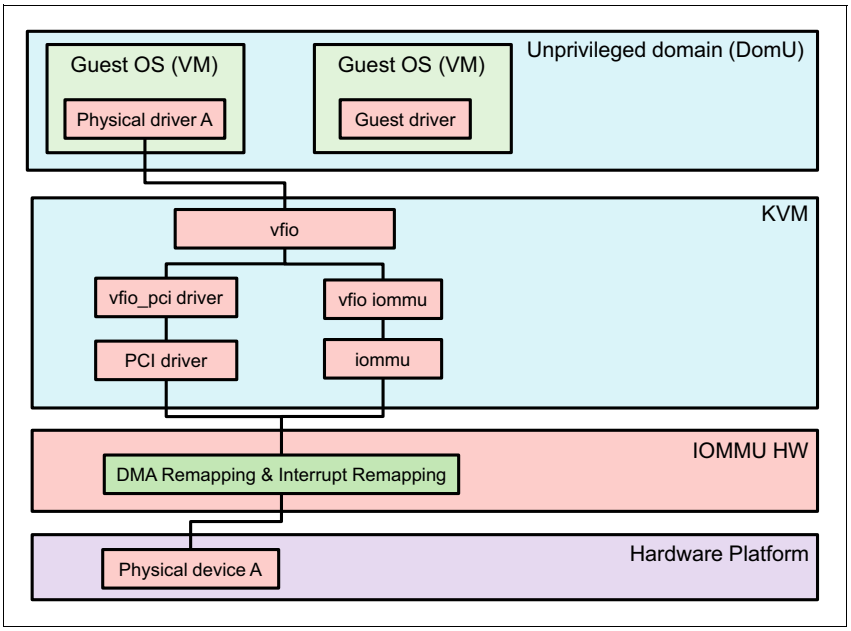

Figure 11 VFIO Framework in Linux Kernel (or KVM)

For more information on VFIO, see

https://www.kernel.org/doc/Documentation/vfio.txt

Default IOMMU modes on Linux OSes

Table 1 shows the default IOMMU operation mode in Linux OSes. The trend indicates that new OSes apply the pass-through mode instead of the DMA translation mode, so the hypervisor of those OSes can provide the direct device access for guest OSes without reconfiguring the default IOMMU mode.

The default IOMMU mode can be changed by appending kernel parameter:

- `iommu.passthrough=0` or `iommu=nopt` (DMA translation)
- `iommu.passthrough=1` or `iommu=pt` (Pass-through)

Table 1 Default IOMMU Operation Mode in Linux OSes

Linux OS	Kernel IOMMU Mode
RHEL 7.x	DMA Translation
RHEL 8.x	Pass-through
SLES12 SP4 or earlier	DMA Translation
SLES12 SP5	Pass-through
SLES15	DMA Translation
SLES15 SP1 or later	Pass-through

Direct Device Access Use Case in Virtualization Environment

This section shows how to attach a direct device access to a guest OS and takes a deep dive into IOMMU status change of the hypervisor via the `crash` utility.

Prerequisites are as follows:

- Enable IOMMU in System Setup (UEFI)
- Enable the IOMMU pass-through mode in Linux kernel via a kernel parameter (refer to “Default IOMMU modes on Linux OSes” on page 13 for detail).

Table 2 shows the server configuration we used in our lab environment.

Table 2 Test Environment

Component	Description
Operating system	RHEL8.2 with the inbox kernel (4.18.0-193.el8.x86_64)
Hardware	Lenovo ThinkSystem™ SR665 with AMD EPYC 7002 “Rome” family of processors

Attach a Direct Device Access to a Guest OS via virt-manager

The following steps show how to directly attach an I/O device to a guest OS so that the I/O requests can be translated by an IOMMU hardware.

1. Launch virt-manager application as shown as Figure 12.

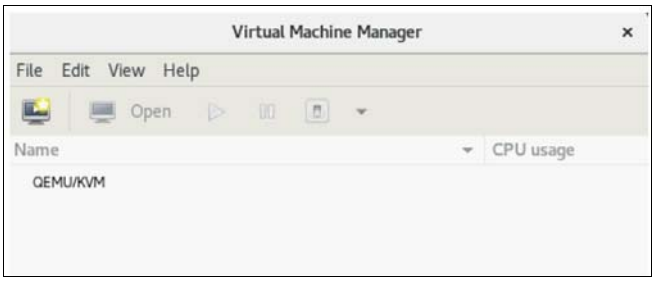

Figure 12 virt-manager application

2. Install a RHEL8.2 guest OS via virt-manager application.
3. After the installation, shut down the guest OS
4. Attach a physical device to the guest OS by clicking **Add Hardware** → **PCI Host Device** and select your network device (Intel I350 in our lab server)’ as shown in Figure 13 and Figure 14.

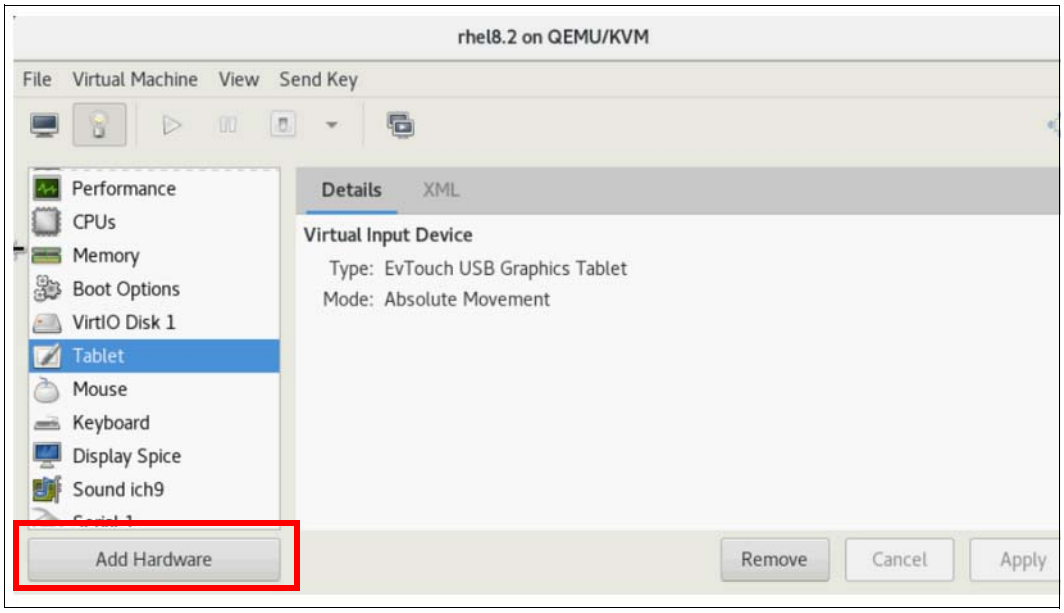

Figure 13 Hardware List in virt-manager

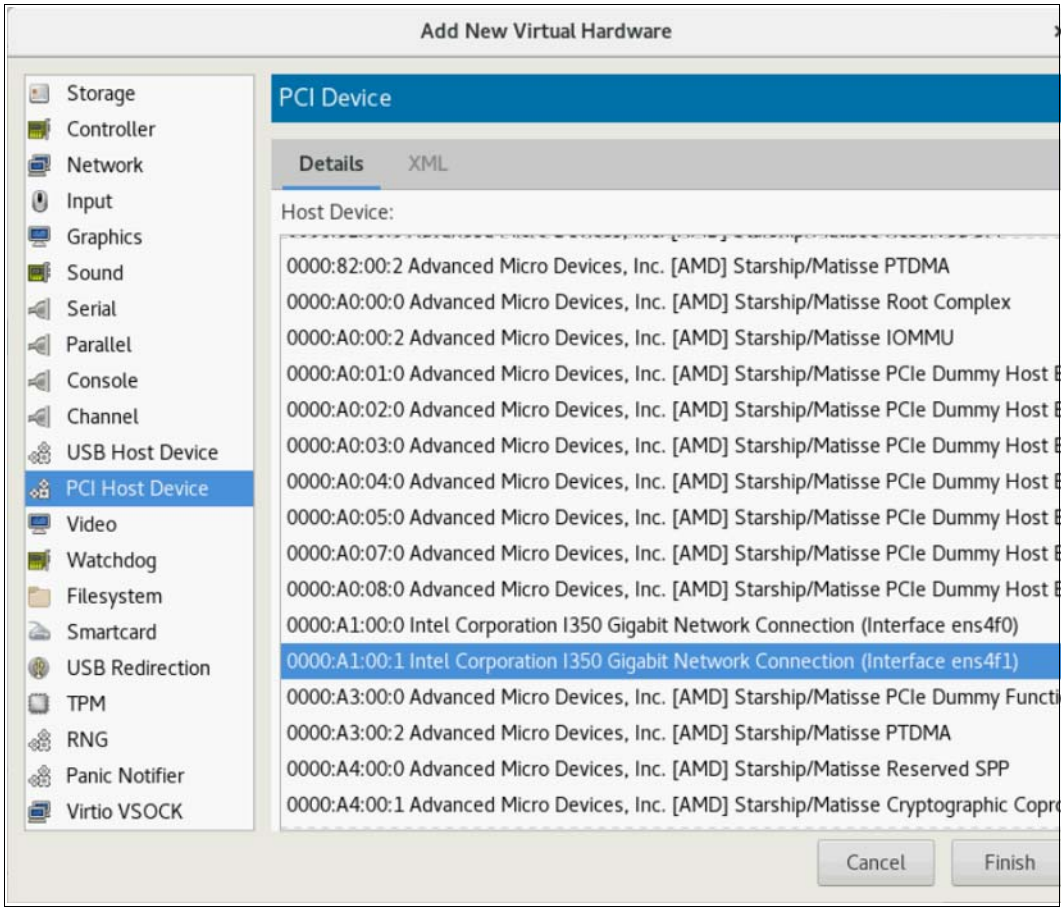

Figure 14 Add a Direct Device Access to a Guest OS

5. Power on the guest OS and check the network device using the `lspci` command as shown in Figure 15. The command output shows that the guest OS is equipped with a direct device access that is exported by the hypervisor.

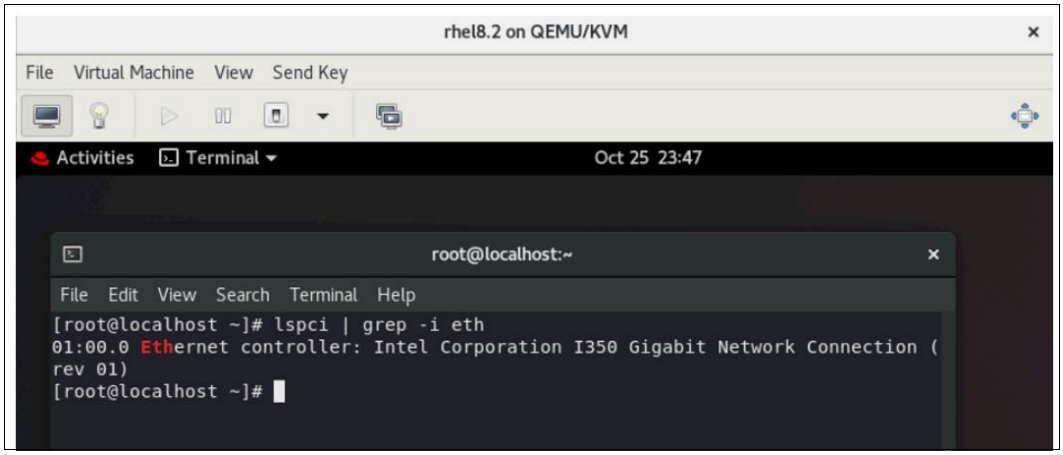

Figure 15 Show the Direct Device in a Guest OS

KVM (hypervisor) Changes Before/After Booting into a Guest OS with a Direct Device Access

Figure 16 shows the IOMMU information of the KVM before booting into a guest OS with a direct device access (a1:00.1 is an Intel I350 network adapter in this example).

```
# lspci | grep -i eth
a1:00.0 Ethernet controller: Intel Corporation I350 Gigabit Network Connection
 (rev 01)
a1:00.1 Ethernet controller: Intel Corporation I350 Gigabit Network Connection
 (rev 01)

# lsmod | grep vfio
<empty>

# dmesg | grep "iommu group" | tail -n 5
[    3.576071] pci 0000:80:08.1: Adding to iommu group 115
[    3.576124] pci 0000:81:00.0: Adding to iommu group 116
[    3.576173] pci 0000:81:00.2: Adding to iommu group 117
[    3.576227] pci 0000:82:00.0: Adding to iommu group 118
[    3.576279] pci 0000:82:00.2: Adding to iommu group 119

[crash utility]
crash> gdb x /4g (amd_iommu_dev_table + 0xa101)
0xffff926d61d42020:     0x6000000000000003      0x0000000000000066
0xffff926d61d42030:     0x2000000819256011      0x0000000000000000
crash> gdb x /4g amd_iommu_pd_alloc_bitmap
0xffff926dc7d58000:     0xffffffffffffffff      0x01ffffffffffffff
0xffff926dc7d58010:     0x0000000000000000      0x0000000000000000
```

Figure 16 Status before Booting into a Guest OS with a Direct Device Access

Observation of the output are as follows:

- The vfio kernel modules are not loaded yet.
- When initializing IOMMU, Linux kernel creates 120 IOMMU groups (0-119).

Use the **crash** utility to observe AMD IOMMU structure:

- The global variable `amd_iommu_dev_table` is the device table described in Section 2.2.2 of AMD I/O Virtualization Technology (IOMMU) Specification. Each entry of the device table (DTE: Device Table Entry) associating with each physical device as well as indexing via a device ID (PCI bus ID/device ID/function ID: BDF) defines the attributes such as address translation, interrupt remapping, and control bits.
  - DTE[51:12]: This field denotes Host Page Table Root Pointer that stores the system physical address of the I/O page table. Figure 16 shows the value is 0x0 (extracted from 0x6000000000000003). It means this device (BDF: a1:00.1) is not ready for address translation.
- The global variable `amd_iommu_pd_alloc_bitmap` denotes a bitmap. Each bit represents the allocated protection domain. The bit position of the global variable is used to configure DTE[79:64] (Domain ID). The bit position '0' is never allocated and its value is always '1'. In this example, the value `0x01ff_ffff_ffff_ffff ffff_ffff_ffff_ffff` shows that 120 protection domains are allocated. This equals to the allocated IOMMU groups.
  - Total bit set: 121
  - Allocated protection domains: 121 – 1 = 120 (The bit position '0' is never allocated).

Figure 17 shows the IOMMU information of the KVM after booting into a guest OS with a direct device access.

```
[root@rh82 ~]# lsmod | grep vfio
vfio_pci               53248  1
vfio_virqfd            16384  1 vfio_pci
vfio_iommu_type1       32768  1
vfio                   36864  5 vfio_iommu_type1,vfio_pci
irqbypass              16384  4 vfio_pci,kvm

[crash utility]
crash> gdb x /4g (amd_iommu_dev_table + 0xa101)
0xffff926d61d42020:     0x6000000688294603      0x0000000000000079
0xffff926d61d42030:     0x2000000819256011      0x0000000000000000
crash> gdb x /4g amd_iommu_pd_alloc_bitmap
0xffff926dc7d58000:     0xffffffffffffffff      0x03ffffffffffffff
0xffff926dc7d58010:     0x0000000000000000      0x0000000000000000
```

Figure 17 Status after Booting into a Guest OS with a Direct Device Access

The observations are as follows:

- The vfio driver are loaded automatically. This observation aligns with Figure 11 on page 13.

Use the **crash** utility to observe the AMD IOMMU structure:

- DTE[51:12] (Host Page Table Root Pointer): It shows a valid value `0x688294000` (extracted from `0x6000000688294603` and set the lowest 12 bits as 0)
- DTE[79:64] (Domain ID): The domain ID is changed from 0x79 to 0x66. 0x79 means the 121st allocated protection domain (requested from vfio drivers). This also reflects the global variable `amd_iommu_pd_alloc_bitmap` whose value is changed from `0x01ff_ffff_ffff_ffff ffff_ffff_ffff_ffff` to `0x03ff_ffff_ffff_ffff ffff_ffff_ffff_ffff` (one newly allocated protection domain).

Summary

IOMMU hardware has been widely adopted within a virtual environment to improve the system performance. This paper describes PCI device virtualization models, IOMMU subsystem in Linux kernel, Linux IOMMU DMA translation mode and pass-through mode, and how to directly use a PCI device in a guest OS.

For more info about IOMMU, see the following articles:

- Utilizing IOMMUs for Virtualization in Linux and Xen

  http://developer.amd.com/wordpress/media/2012/10/IOMMU-ben-yehuda.pdf

- Configuring A Host For PCI Passthrough

  https://access.redhat.com/documentation/en-us/red_hat_virtualization/4.0/html/installation_guide/appe-configuring_a_hypervisor_host_for_pci_passthrough

Acronyms

ACPI	Advanced Configuration and Power Interface
AMD-Vi	AMD I/O Virtualization Technology
BDF	Bus Number/Device Number/Function Number
DMA	Direct Memory Access
DMAR	DMA Remapping Reporting
DTE	Device Table Entry
GCR3	Guest Control Register 3
GPA	Guest Physical Address
GVA	Guest Virtual Address
IOMMU	Input Output Memory Management Unit
IVDB	I/O Virtualization Definition Block
IVHD	I/O Virtualization Hardware Definition
IVMD	I/O Virtualization Memory Definition
IVRS	I/O Virtualization Reporting Structure
MMU	Memory Management Unit
SPA	System Physical Address
VMM	Virtual Machine Monitor
VT-d	Intel Virtualization Technology for Directed I/O

References

See these web resources for more information:

- Linux virtualization and PCI passthrough

  https://developer.ibm.com/tutorials/l-pci-passthrough

- AMD I/O Virtualization Technology (IOMMU) Specification

  https://www.amd.com/system/files/TechDocs/48882_IOMMU_3.05_PUB.pdf

- Intel Virtualization Technology for Directed I/O Architecture Specification

  https://software.intel.com/content/www/us/en/develop/download/intel-virtualization-technology-for-directed-io-architecture-specification.html

- Virtual Function I/O (VFIO)

  https://www.kernel.org/doc/Documentation/vfio.txt
- VFIO driver analysis

  https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2019/08/21/vfio-driver-analysis

Author

Adrian Huang is a Senior Linux Engineer in the Lenovo Infrastructure Solutions Group based in Taipei, Taiwan. He has experience with Linux kernel IOMMU subsystem, block device layer and memory management. He also contributes kernel patches to kernel community.

Special thanks to the following people for their contributions and suggestions:

- Xiaochun, Lenovo Linux Engineer
- Song Shang, Lenovo Linux Engineer
- Gary Cudak, Lenovo OS Architect
- David Watts, Lenovo Press

Notices

Lenovo may not offer the products, services, or features discussed in this document in all countries. Consult your local Lenovo representative for information on the products and services currently available in your area. Any reference to a Lenovo product, program, or service is not intended to state or imply that only that Lenovo product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any Lenovo intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any other product, program, or service.

Lenovo may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

Lenovo (United States), Inc.
1009 Think Place - Building One
Morrisville, NC 27560
U.S.A.
Attention: Lenovo Director of Licensing

LENOVO PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. Lenovo may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

The products described in this document are not intended for use in implantation or other life support applications where malfunction may result in injury or death to persons. The information contained in this document does not affect or change Lenovo product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of Lenovo or third parties. All information contained in this document was obtained in specific environments and is presented as an illustration. The result obtained in other operating environments may vary.

Lenovo may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any references in this publication to non-Lenovo Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this Lenovo product, and use of those Web sites is at your own risk.

Any performance data contained herein was determined in a controlled environment. Therefore, the result obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

This document was created or updated on April 20, 2021.

Send us your comments via the **Rate & Provide Feedback** form found at
http://lenovopress.com/lp1467

Trademarks

Lenovo and the Lenovo logo are trademarks or registered trademarks of Lenovo in the United States, other countries, or both. These and other Lenovo trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by Lenovo at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of Lenovo trademarks is available from https://www.lenovo.com/us/en/legal/copytrade/.

The following terms are trademarks of Lenovo in the United States, other countries, or both:

Lenovo® Lenovo(logo)® ThinkSystem™

The following terms are trademarks of other companies:

Intel, and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.