Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Investigating Effects of Hardware Isolation in High-Speed Network Environments

Simon Ellmann

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Investigating Effects of Hardware Isolation in High-Speed Network Environments

Auswirkungen von Hardware-Isolation in Hochgeschwindigkeits-Netzwerkumgebungen

Author:	Simon Ellmann
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Paul Emmerich, M. Sc.
	Benedikt Jaeger, M. Sc.
	Florian Wiedner, M. Sc.
Date:	April 15, 2021

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, April 15, 2021

Location, Date

Signature

Abstract

Hardware isolation plays an important role in today's computer systems. Internet companies and telecommunication providers rely on it for their heavily virtualized infrastructures with software-defined networking, network function virtualization, containers and virtual machines, and consumer hardware uses it to protect itself against malicious or faulty external peripherals.

One common way to isolate hardware is the use of input-output memory management units (IOMMUs), multipurpose devices to virtualize I/O memory that can be found in all kinds of devices nowadays, from high-end servers to mobile devices like Apple's iPhones, and are inevitable for fast I/O on platforms like Amazon's Elastic Cloud Computing. Due to the widespread use of IOMMUs, questions arise to which extent IOMMUs impact performance and safety/security of the systems they are used in.

We address these questions in the area of high-speed network environments. For our analyses, we use `ixy.rs`, a state-of-the-art user space network driver for Intel's 82599 network cards written in Rust. We show that IOMMUs have a minor impact on performance in most cases. In some cases, however, IOMMUs lead to a loss of packet throughput of more than 50%. While our suspicion of a side-channel vulnerability in the translation lookaside buffers of IOMMUs is not confirmed, we note that IOMMU protection against malicious peripherals is weak.

Our contributions are a new driver to `ixy.rs` for virtual functions, support for legacy Intel IOMMUs with limited second-level address translation capabilities, and a tool to perform accurate timing measurements on network card memory operations. Besides, we have fixed an error in DPDK's ixgbevf driver.

Zusammenfassung

Hardware-Isolation spielt in heutigen Computersystemen eine wichtige Rolle. Internetfirmen und Telekommunikationsanbieter sind mit ihrer stark virtualisierten Infrastruktur auf Hardware-Isolation angewiesen, und Verbraucherprodukte nutzen sie, um sich vor böswilligen oder fehlerhaften externen Geräten zu schützen.

Eine gängige Methode zur Isolierung von Hardware ist die Verwendung von IOMMUs, Mehrzweckgeräten zur Virtualisierung von IO-Speicher, die heutzutage in allen Arten von Geräten zu finden sind, von High-End-Servern bis hin zu mobilen Geräten wie Apples iPhones, und die für schnelle IO-Operationen auf Plattformen wie Amazons Elastic Cloud Computing unumgänglich sind. Aufgrund der weiten Verbreitung von IOMMUs stellt sich die Frage, inwiefern sie die Leistung und die Sicherheit der Systeme, in denen sie eingesetzt werden, beeinflussen.

Wir adressieren diese Frage im Bereich von Hochgeschwindigkeits-Netzwerkumgebungen. Für unsere Analyse verwenden wir `ixy.rs`, einen in Rust geschriebenen, modernen User-Space-Netzwerktreiber für Intels 82599-Netzwerkkarten. Wir zeigen, dass IOMMUs in den meisten Fällen einen geringen Einfluss auf die Performance haben. In manchen Fällen jedoch führen IOMMUs zu einem Performanceverlust von mehr als 50%. Während sich unser Verdacht auf eine Seitenkanal-Schwachstelle in den Übersetzungspuffern von IOMMUs nicht bestätigt, stellen wir fest, dass der Schutz von IOMMUs gegenüber bösartigen Geräten gering ist.

Mit dieser Arbeit tragen wir einen neuen Treiber zu `ixy.rs` für virtuelle Funktionen, Unterstützung für ältere Intel-IOMMUs mit eingeschränkter Second-Level-Adressübersetzung und ein Werkzeug zur Durchführung genauer Zeit-Messungen auf Speicherzugriffe von Netzwerkkarten bei. Des Weiteren wurde ein Fehler in DPDKs ixgbevf-Treiber behoben.

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Dr.-Ing. Georg Carle, for his enduring support and confidence in my abilities, for the opportunity to conduct research at his chair, for our discussions on my work, and for enabling me to participate in conferences and workshops.

I would like to say a special thank you to Paul Emmerich for all the pleasurable conversations, his moral support, and the invaluable feedback on my work. His profound knowledge and enthusiasm for research made my time at the chair all the more enjoyable.

I also wish to thank Florian Wiedner and Benedikt Jaeger for their guidance and feedback, which contributed significantly to the success of this thesis.

Furthermore, I would like to thank Heiko Simoneit, Catharina Großmaß and Clemens Horn for their friendship, unclouded optimism in all circumstances and our leisure activities.

And last but not least, I would like to express my deep gratitude to my family for their unconditional support throughout the last years.

Contents

Chapter 1

Introduction

High-speed network environments have seen a major shift towards virtualization and cloud computing in the last two decades. This development began in the early 2000s, when Internet giants like Amazon, Google and Facebook had to cope with rapid company growth and a sharp expansion of their infrastructure. By deploying virtualization techniques on their servers, these companies managed to create scalable platforms for their services while utilizing existing hardware resources more efficiently.

Virtualization in this context does not necessarily refer to hardware virtualization or virtual machines. Various solutions were used by the big companies to abstract and unify their computing environments, from virtual machines to custom software for storage/network virtualization or operating system (OS)-level virtualization like containers later on. Google, for example, is known to have dispensed with classic virtualization in the form of hypervisor and virtual machines on its platforms early on. For its cluster management software Borg, Google preferred custom solutions and eventually containerization due to the relatively high costs of classic virtualization [39].

Telecommunication providers have also begun to use virtualization techniques on a broader basis. Through SDN, providers such as Orange, Verizon and Deutsche Telekom [9] have created more flexible, centrally managed networks. Google uses SDN on a large scale, e.g., for its B4 datacenter WAN [23]. Another virtualization technique employed by telecommunication providers and IT companies is NFV, i.e., virtualization of functionality traditionally implemented in dedicated hardware like NAT, load balancers or firewalls [8]. SDN and NFV offer several benefits to these enterprises. With SDN, networks can be created and orchestrated remotely, services can be chained on-the-fly, and providers can mitigate traffic bursts by dynamically adapting routes and bandwidths.

With NFV, dedicated middle-boxes can be reduced and replaced by off-the-shelf hardware, and manual work is no longer needed to adapt network functionality.

A side effect of large-scale deployment of virtualization techniques by the large IT companies was the invention of cloud computing. Like virtualization, cloud computing is primarily a cost-saving measure: Companies rent out their scalable platforms to other companies and end users to maximize utilization of their infrastructure. Major companies use cloud computing as part of their business models, namely Amazon with AWS since 2006, Microsoft with Microsoft Azure since 2010 and Google with the Google Compute Engine since 2012.

Cloud computing offers a wide range of possible applications and is an active subject of research by companies and universities. Topics of interests are, for example, secure computing in the cloud, large scale data analysis or load balancing.

While not necessarily being built on classic virtualization technique themselves, Amazon AWS, Microsoft Azure and Google Compute Engine offer virtual machines to customers. For high-performance applications, features like Amazon's "Enhanced Networking" are available which mitigate one of the limiting factors in virtualized environments: poor I/O performance. In order to achieve high throughput rates and low latency, direct pass-through of devices to virtualized environments is essential. However, as hardware in current computer systems is generally considered trustworthy and I/O devices have unrestricted access to host memory by design, devices cannot be passed-through in a safe manner as they are since software in virtual environments could take over its host by abusing the device's memory access privileges.

The fact that I/O devices have unrestricted access to memory is a consequence of historic developments. In the earliest computers, data transfer between memory, CPU and peripherals was done by the CPU. However, CPUs are not particularly good at copying data, and I/O operations are expensive. To reduce the time spent on memory transactions by the CPU, direct memory access (DMA) was introduced. With DMA, data transfers between I/O devices and memory were initially performed by a separate controller (third-party DMA) and later by the I/O devices themselves (first-party DMA). As the term implies, devices that use DMA address memory directly, hence have unrestricted access to host memory. This technique is still employed by I/O devices today, first and foremost by PCIe devices.

For fast pass-through of I/O devices with real hardware isolation, computer manufacturers included a new component in their systems called input-output memory management unit (IOMMU). Similar to memory management units (MMUs), IOMMUs introduced virtual memory for I/O devices and a mechanism to restrict DMA accesses of peripherals

to the virtualized memory. With such an IOMMU that can only be configured by the hypervisor or the operating system of the host, pass-through can be performed in a safe manner.

In fact, IOMMUs or their predecessors were used before the advent of I/O virtualization technology on x86 in 2008. Some of the first machines with an IOMMU were the SPARC stations of the Sun-4 architecture launched in 1987. As early bus protocols at times supported smaller address spaces than the hosts they were used in, it is suspected that these early IOMMUs were included into the SPARC computers to improve I/O performance by avoiding bounce buffers, i.e., expensive copying of memory between DMA-able memory and buffers in non-DMA-able memory [38, pp. 28 ff.]. Another early IOMMU was the Graphics Address Remapping Table (GART), used for graphic cards to provide a contiguous view of host memory.

In 2005/2006, Intel (VT-x) and AMD (AMD-V) released new processor extensions to support hardware virtualization. Slow I/O performance of early Intel VT-x processors [1] was remedied by the first CPUs employing Intel VT-d in 2008, i.e., Intel's I/O virtualization features including an IOMMU, respectively with AMD-V in 2011 by AMD's FX CPUs. Although primarily intended for virtualization, these IOMMUs were created as multipurpose devices.

Usage of IOMMUs has changed. In recent years, protection against malicious or faulty I/O devices moves to the forefront. Necessity for protection in this domain was already apparent in the early 2000s, when attacks through externally connected DMA-able I/O devices, e.g., using Apple's FireWire, were carried out [4]. With the mass adoption of USB-C and Thunderbolt (another DMA-capable protocol), the topic is gaining new importance.

Nowadays, IOMMUs are not only used in servers and desktop computers but also mobiles devices like smartphones. Some ARM based Android phones are shipped with IOMMUs, and Apple has developed its own variant for its iPhones to protect the WiFi stack [7].

In summary, we note that IOMMUs today are used for various purposes, from virtualization in server architectures to protection against malicious peripherals in consumer hardware. Due to their versatiliy and prevalence, IOMMUs represent an interesting object of research. In this thesis, we investigate performance and safety/security impacts of IOMMUs in high-speed network environments with a focus on Intel's and AMD's IOMMUs on x86.

Chapter 2

Background

I/O devices interact with the CPU via interrupts, registers and shared memory. We present an overview of how these communication primitives work with IOMMUs in high-speed network environments in the following sections, focusing on PCIe, NICs, IOMMUs on Linux, and SR-IOV.

2.1 Overview

On modern systems, peripherals, processor and main memory are interconnected via a wide set of connection standards and protocols. Devices are either directly attached to the CPU via PCIe or to an I/O hub known as Platform Controller Hub (Intel) or chipset (AMD). Direct connections are preferred for I/O devices with high performance requirements such as video cards or NVMe controllers. On enterprise hardware, e.g.,

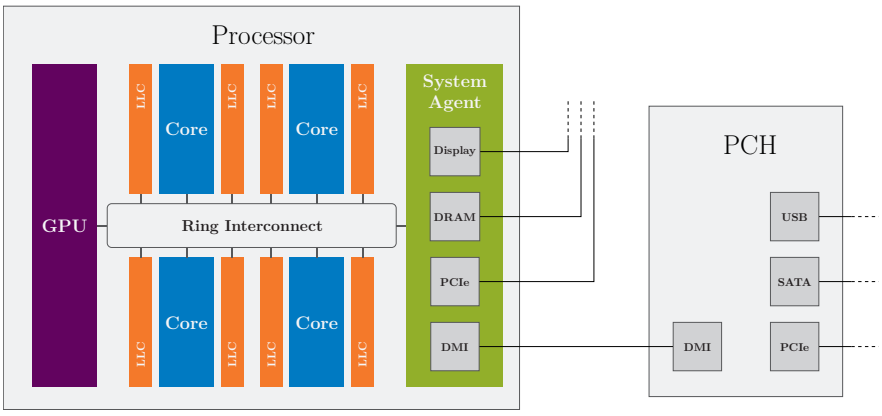


Figure 2.1: Processor and Platform Controller Hub (PCH) on an Intel system.

servers, most PCIe devices are directly attached to the CPU, whereas slower peripherals like on-board Gigabit network cards are connected to the I/O hub. On consumer hardware, more devices are usually attached to the I/O hub. The I/O hub resides on a separate chip and is connected to the CPU via Direct Media Interface (Intel) or Unified Media Interface (AMD). Figure 2.1 illustrates the relationship between CPU, I/O hub and peripherals. Conceptually, system agent and Platform Controller Hub (PCH) depicted in Figure 2.1 are successors of northbridge and southbridge from previous chipsets. While memory controller and other northbridge functions were incorporated into the CPU as so-called system agent, southbridge and remaining northbridge functions were moved to the PCH.

2.2 PCI Express

As de-facto standard, Peripheral Component Interconnect Express (PCIe) is used for communication with most peripherals. PCIe is a high-speed serial bus that was designed to replace PCI and PCI-X. Unlike the former, PCIe is based on a point-to-point topology where all communication data is encapsulated in PCIe packets, and supports hotplugging of devices. Conceptually, PCIe consists of various communication layers and – similar to TCP – provides mechanisms like flow control and acknowledgement messages to ensure reliable data transmission.

Physically, PCIe devices are connected via lanes which consist of two differential pairs each, i.e., four wires per lane. The two pairs of PCIe lanes are used unidirectionally as one transmit and one receive channel. Together, they form a bidirectional link. For interrupts, no physical pin is used. Instead, in-band messages signal interrupts to the interrupt controller which in turn interrupts the CPU. To increase throughput, devices may use up to 32 lanes. With today's commonly used PCIe 4.0 (which was released in 2017), a serial data rate of 16 Gbit/s per lane or 1.97 GB/s (taking the 128b/130b line code into account) can be achieved [36]. Compared to PCIe 1.0 (released in 2003), throughput of the lanes has increased by a factor of eight.

Between PCIe devices and processor, switches may be used. Figure 2.2 depicts a typical PCIe tree. On the CPU side, connections end in the PCIe root complex which is part of the system agent on Intel systems and the bridge between CPU, main memory and devices. On the device side, connections end at PCIe endpoints. PCIe devices may initiate transfers to main memory through the PCIe root complex, acting as DMA bus masters that access main memory independently of the CPU.

PCIe uses three layers for communication: a physical layer, a data link layer and a transaction layer. The physical layer of PCIe is responsible for link initialization and

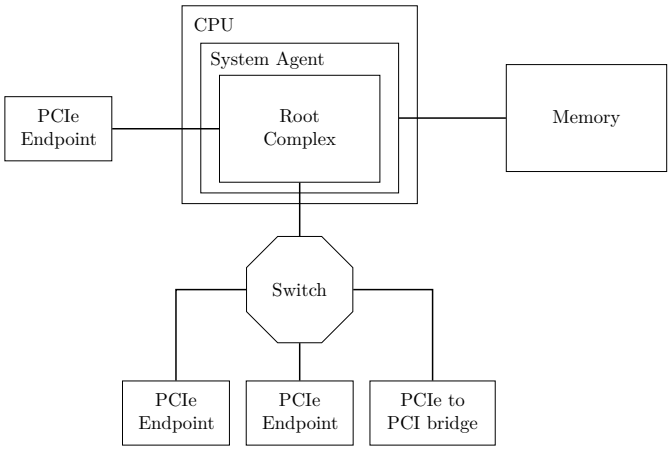


Figure 2.2: Exemplary PCIe topology on an Intel system.

point-to-point data transfer. The data link layer ensures reliable transport of data between individual PCIe endpoints, or endpoints and the root complex. The transaction layer contains user application data or configuration data for the link, and routing information.

Every PCIe transmission consists of four bytes that mark the start of a new packet [36, pp. 238 ff.], a header of 12 B (or 16 when using 64-bit addresses) [36, p. 98], a payload of up to 4096 B [36, p. 729] (depending on the maximum payload size on the link), and 4 to 8 B of packet checksums [36, p. 173]. Thus, the minimal packet overhead for headers, checksums, etc. is 20 B.

PCIe packets are categorized into requests and completions [36, p. 72]. Requests are memory, I/O or configuration reads and writes (to the PCI configuration space) that are either posted, i.e., expect a completion (reply), or non-posted. The PCI configuration space is a standardized register space that contains information about a PCI(e) device such as vendor and device ID, device class or memory access type and addresses of the device's registers (stored in the Base Address Registers or BARs). Among others, memory space is used by a device to access main memory and by the CPU to access device registers. I/O space is included for backwards compatibility and will probably be deprecated in the future [36, p. 71].

Device registers can be accessed via memory-mapped I/O or port-mapped I/O. With memory-mapped I/O, main memory and I/O devices share the same address space, i.e., all physical addresses refer to devices or main memory. With legacy port-mapped I/O, special CPU instructions and an I/O pin on the CPU or a separate I/O bus are used to perform I/O.

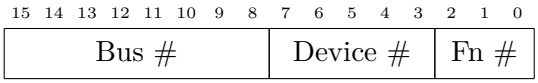


Figure 2.3: PCIe IDs as bus/device/function triple.

PCIe uses different schemes to route packets. For memory and I/O requests, 32 to 64 bit addresses are used. Requests to the configuration space are routed by an ID consisting of an eight bit bus number, five bit device number and three bit function number [36, p. 591] (commonly referred to as bus/device/function (BDF) number). See Figure 2.3 for reference. PCIe also specifies an "Alternative Routing-ID" format where the 16-bit PCIe ID is split in an 8-bit bus number and an 8-bit function number, i.e., device and function number merge into one [36, p. 591]. However, this scheme does not yet seem to be used in practice [38, p. 24].

On system boot, every PCIe root complex is enumerated to determine which slots have devices installed. During enumeration, the BIOS/UEFI or OS probe the PCI configuration space for all possible combinations of bus and device number, i.e., by reading vendor and device ID of PCIe switches and endpoints. It suffices to probe for function 0 for every combination of bus and device number to determine a device's presence since every device is required to implement function 0. For every detected device, the maximum payload size on the link is determined and the device is assigned its respective BDF. Bus and device numbers might be re-assigned by the enumeration software if a PCIe device is hotplugged.

After a device's presence has been detected, every base address register field in the configuration space is read to determine the address space needed to access the registers of the device and subsequently every device is assigned a portion of the host physical address space. If software sets one of the device's registers through memory-mapped I/O, e.g., during device setup by a device driver, a memory write to the physical address is issued. The PCIe root complex – which unites the physical address space of all its devices – receives the memory write, and initiates a PCIe memory write transaction to the device.

2.3 Network Interface Controllers

A common kind of PCIe peripherals are network cards used for communication between computer systems via wireless (e.g., WLAN) or wired (e.g., LAN) networks, often connecting hosts via a gateway to the Internet. Another term for network card is network interface controller (NIC).

Manufacturers produce NICs for a wide range of applications, be it for data centers with high performance requirements, or mobile phones and desktop computers of the consumer sector. As versatile as these NICs are, so are their drivers, and complexity of drivers and devices has increased over the years: whereas early NICs were solely capable of receiving and transmitting network packets, modern network cards offer plenty of (hardware-offloading) features like checksum calculations, encryption and authentication of packets, VLAN tagging and flow control, time syncing, traffic shaping, etc.

Complexity of NICs and drivers is reflected in several ways. On the one hand in manuals (the datasheet for network cards of the Intel 82599 family [20] consists of more than 1,000 pages) and driver size (exceeding 100,000 lines of code for some devices [15]). On the other hand in devices like Intel's XL710 which follow a more firmware-driven design where much functionality is executed on the network card, and driver and NIC communicate via a message based interface [16]. The downside of this complexity in the currently dominant monolithic operating systems is a lack of reliability, safety and security: in case of Linux, 39 out of 40 memory bugs found in the kernel in 2017 were located in device drivers [15].

Although NICs have changed a lot in the last decades, most of them still use a seemingly simplistic interface to communicate reception or transmission of packets: descriptor rings. Descriptor rings are circular buffers that contain information about network packets and are shared between NIC and host. NICs that use descriptor rings have RX and TX descriptor rings. Figure 2.4 shows a TX descriptor ring. Descriptors in TX descriptor rings describe outgoing packets, i.e., size of the packet to be sent, memory location of packet data (e.g., of buffers in a memory pool), whether offloading features like TCP/UDP checksum insertion should be applied by the NIC, whether transmission of the packet should be reported through descriptor writeback, etc. Conversely, descriptors in RX descriptor rings describe incoming packets, e.g., where packet data of received packets can be stored by the NIC.

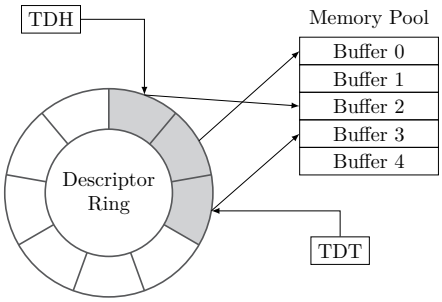


Figure 2.4: TX descriptor ring of a Network Interface Controller.

Ownership of descriptor rings is shared between NIC and host through two queue pointers, a head and a tail pointer. In case of the TX descriptor ring, the NIC manages the head pointer (TDH) while the device driver manages the tail pointer (TDT). At device initialization, head and tail pointer are set to the first descriptor. Once a packet is queued by the driver, i.e., the descriptor in the TX descriptor ring has been changed appropriately, the driver updates the tail pointer to the next descriptor, thereby transferring ownership of all descriptors between head and tail pointer to the NIC. Conversely, the NIC updates the head pointer once a descriptor was processed and the packet data has been fetched from main memory by the NIC.

If the tail pointer points to the descriptor preceding the head pointer's descriptor, the TX queue is full and no new data can be queued for transmission. Consequently, the driver has to wait until some packets have been sent out by the NIC.

2.4 Direct Memory Access

Access to main memory by hardware without CPU involvement is known as direct memory access (DMA). By bypassing the CPU, expensive memory operations can be offloaded, potentially improving performance as the CPU is no longer involved in all memory transactions and therefore able to perform other operations while data from/to main memory is transferred. In multi-core processors, DMA is sometimes used to transfer data between cores [24]. In conjunction with PCIe devices, DMA is performed through PCIe memory read and write transactions.

Using PCIe, no separate DMA controller is needed (third-party DMA), instead reads and writes are issued directly from bus masters (first-party DMA), i.e., PCIe devices allowed to read/write memory or I/O space. Memory reads and writes of main memory travel up the PCIe tree to the root complex and subsequently to main memory via the memory controller. Typically, physical addresses are used to access main memory in the same way they are used by the CPU when accessing device registers through memory-mapped I/O. However, use of physical addresses is troublesome as PCIe devices may read from or write to any memory address of main memory, which is a problem for

- device pass-through to virtual machines, since virtual machines might read host memory outside of their access range through the memory accessing capabilities of PCIe devices;
- legacy 32-bit devices on 64-bit hosts, since they cannot access memory outside of their address space – i.e., the first 4 GiB of host physical address space – without bounce buffers and expensive memory copying by the OS;

- avoiding harm from malicious or faulty drivers, since secrets can be read from main memory and OS data structures can be corrupted;
- avoiding harm from malicious or faulty devices;
- providing secure unprivileged access to hardware, e.g., in case of user space network drivers.

To solve these problems, virtual addresses are used for I/O devices on modern computer systems instead of physical addresses. Similar to process virtual addresses and MMUs, I/O virtual addresses (IOVAs) are assigned to devices and a translation unit is inserted into the data path that restricts memory accesses to a device's memory region. Such translation units are commonly known as input-output memory management units (IOMMUs).

2.5 Input-Output Memory Management Units

IOMMUs are multipurpose devices that primarily virtualize the memory space of peripherals and provide interrupt remapping. Although there is a trend to use IOMMUs as a protection mechanism against malicious or faulty peripherals, the primary application for IOMMUs is and remains virtualization, i.e., direct pass-through of I/O devices to virtual machines for higher throughput and lower latency.

Conceptually, IOMMUs may be used for any kind of I/O devices and any kind of connection standard. In practice, IOMMUs are implemented in the PCIe root complex since most if not all I/O devices in today's systems are connected to the CPU via the PCIe root complex: modern interfaces like Thunderbolt, NVMe and SATA Express are directly built on top of PCIe, and for older connection standards like USB or I^2C, PCIe hubs are used. The focus of IOMMUs on PCIe is also reflected in code like Linux's IOMMU API that is closely centered around PCIe instead of providing an abstract interface (e.g., focusing on PCIe BDF instead of generic groups).

Vendors use different marketing names for their IOMMU solutions. Intel calls it "Intel Virtualization Technology for Directed I/O" (VT-d), AMD uses the term "AMD I/O Virtualization Technology" (AMD-V), and ARM does without such a designation, simply denoting its IOMMUs as system memory management units (SMMUs).

In the VT-d specification, Intel describes different virtualization models for I/O devices, like direct assignment to virtual machines and I/O device sharing (see SR-IOV in Section 2.7), as well as the capabilities provided by Intel VT-d required for these models, namely I/O device assignment, DMA remapping, interrupt remapping and "reliability"

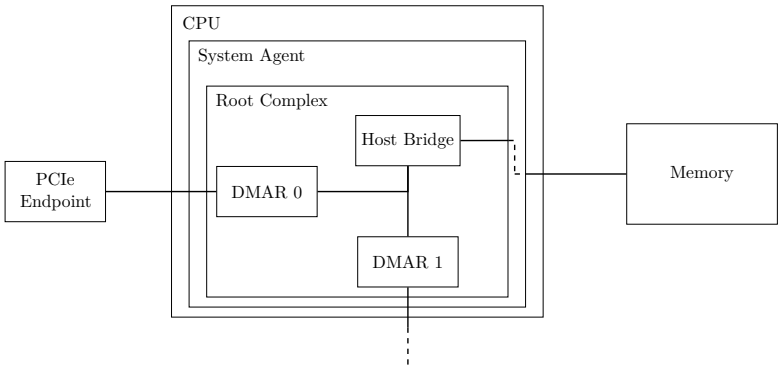


Figure 2.5: DMAR units in the PCIe root complex of an Intel system.

features. Besides virtualization, Intel also suggests some intra-OS use cases for DMA remapping: protection of critical OS data from I/O devices, isolation of DMA accesses, support of high memory addresses for legacy I/O devices.

DMA remapping, i.e., address translation and access validation, is executed by the remapping hardware inside of the IOMMU. Intel calls this hardware "DMA Remapping Hardware", AMD uses the term "I/O Virtualization Hardware", and ARM omits an additional term for this part of the hardware. In the following paragraphs, we will use "DMAR units" when referring to the remapping hardware.

Every PCIe root complex may contain one or multiple IOMMUs where each DMAR unit is responsible for a distinct set of I/O buses (Figure 2.5). Devices on IOMMU-managed buses belong to protection domains, i.e., sets of address mappings and access permissions. While domains may consist of multiple devices, devices belong to exactly one domain.

Data structures used for address translation by DMAR units are kept in main memory to be accessible for OS and IOMMUs. In case of multiple IOMMUs, DMAR units may share the same data structures. On IOMMU initialization, the OS sets up these data structures by creating domains and address mappings for the system's I/O devices. Although this happens at system boot, mappings are not fix: the device-to-domain mapping may change (e.g., when a device is assigned to a virtual machine) and address mappings may be altered by the OS (e.g., when DMA memory is mapped or unmapped).

When I/O devices attempt to access main memory, the request is intercepted by the DMAR unit, checked for admissibility and translated appropriately. This procedure consists of two major steps: First, a device's domain is determined by its BDF number. Second, the domain's translation tables are consulted to map the requested address to the host physical address, taking into account the domain's translation scheme. De-

pending on the usage model of the IOMMU, the addresses used by I/O devices may be one of the following:

- IOVAs managed by software on the host,
- guest IOVAs managed by software in a virtual machine,
- guest physical addresses of a virtual machine,
- guest virtual addresses of software running in a virtual machine.

For different address types, hardware implementations of IOMMUs may support different translation schemes. In case of virtualization, IOMMUs may be used for nested translation of addresses: a first-level translation to translate a guest virtual address to a guest physical address, followed by a second-level translation translating the guest physical address to the host physical address [22, pp. 43 ff.].

Figure 2.6 depicts the address translation mechanism of an Intel IOMMU for 48-bit IOVAs to 4 KiB pages. After an attempt to read or write main memory has been intercepted by the IOMMU, the first table to be consulted – if translation was enabled in the global command register of the IOMMU – is the root table. The physical address of the root table is stored in the Root Table Address Register (RTAR) of the IOMMU. The root table consists of 256 root entries 16 B each, i.e., 4 KiB in total. Every root entry has a bit that indicates presence of a context table for this entry. The root table is indexed by the bus number of the device and – in conjunction with device and function number – determines the context entry for this device.

Like the root table, the context table is a 4 KiB page with 256 16 B entries. Every context entry represents an IOMMU domain. Context entries contain a field indicating whether requests belonging to this domain may pass the IOMMU untranslated, and –

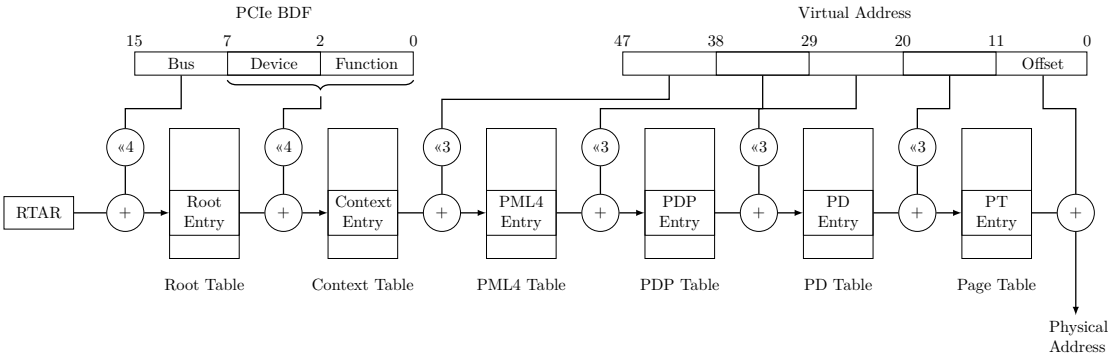


Figure 2.6: Intel IOMMU address translation of 48-bit addresses to 4 KiB pages. Adapted from [33].

in the more likely case that addresses must be translated – a pointer to the first table to be used for address translation for this domain, the PML4 table.

Subsequently, PML4, PDP, PD and the page table are walked by the IOMMU to determine the host physical address belonging to the IOVA. Every table is a 4 KiB page with 512 8 B entries. If there is no translation for a requested address or the access rights for the associated page do not permit access, translation faults and the request is rejected.

In case of 2 MiB pages, no PT table is used for translation, and the last 21 bit of the IOVA represent the offset into the 2 MiB page. In case of 1 GiB pages, no PT and no PD table are used, and the last 30 bit of the IOVA represent the offset into the 1 GiB page. PML4, PDP, PD and PT table form a 4-level translation structure. For 57-bit IOVAs and 4 KiB pages, a 5-level structure is used with an additional PML5 table. For 39-bit IOVAs and 4 KiB pages, only three tables are used.

Intel's IOMMU uses two different formats for the page table entries. For first-level translation, entries equal the format used by the MMU such that the CPU page tables can be re-used for address translation. For second-level translation, a unique format is used.

AMD's IOMMU is mostly identical to Intel's. One of the major differences is that the AMD IOMMU uses one table (called device table) to determine the IOMMU domain from the PCIe BDF identifier while Intel uses two, root table and context table. With 65,536 possible BDF combinations, AMD's device table consists of up to 65,536 entries with 256 bit each, i.e., up to 2 MiB. Another difference of AMD's IOMMU is consistent usage of the MMU page table entries format for all translations.

ARM on the other hand uses a completely different design for its IOMMU. Its SMMUv2 specification describes a register-based architecture for small-scale systems, and SMMUv3 an architecture with IOMMU configuration stored in memory for larger systems. The translation tables of these IOMMUs are compatible to the ARMv8 architecture. Like AMD's and Intel's IOMMU, ARM determines the context of the supplied address before address translation, however, context does not depend on the BDF but a stream ID that is transmitted with every memory transaction.

Common to all IOMMUs is caching of translation information in a context cache and a TLB, the I/O translation lookaside buffer (IOTLB), to speed up address translation. While context cache and IOTLB increase performance, they also increase complexity as caches have to be kept up-to-date and entries must be purged from the caches when they become invalid. If not implemented properly, this creates new vulnerabilities as previous attacks have shown.

2.6 IOMMUs on Linux

Linux supports IOMMUs of all major vendors: besides Intel and AMD, the kernel also includes drivers for IOMMUs from ARM, Qualcomm, Texas Instruments, Samsung, Nvidia, IBM and other, less well-known manufacturers.

IOMMUs are not used by default on all architectures. In case of Intel x86, Linux ignores any DMAR units and enables its own software IOMMU called SWIOTLB. The SWIOTLB is a poor man's IOMMU that uses bounce buffers: At boot time, a contiguous chunk of memory (usually 64 MiB) is reserved in the lower address space of main memory and used for DMA-capable devices with limited addressing capabilities, e.g., legacy 32-bit devices. When an already allocated DMA buffer is to be used by a device that cannot address the buffer directly, memory in the lower address space, the so-called aperture, is linked to that buffer, passed to the device, and data is copied back and forth by the OS between the device-used memory in the aperture and the device-inaccessible buffer.

On AMD systems, IOMMUs are enabled by default. For Intel, this behavior can be enforced through kernel boot parameters. The reasons why Intel's IOMMUs are not enabled by default are not clear. Intel's IOMMUs were disabled for the first time in 2009 due to "recurring issues with DMAR support on certain platforms" [31]. Since then, Linux distributions as Debian and Arch Linux have repeatedly enabled and disabled Intel IOMMU support [17, 19].

When IOMMUs are enabled, Linux checks the ACPI tables of BIOS/UEFI for DMAR units on system boot. For every DMA Remapping Hardware Unit Definition (DRHD) in the DMAR reporting ACPI tables, Linux initializes the DMAR unit through its respective device driver, creates a default IOMMU domain and assigns all PCIe devices to IOMMU groups, where each group represents the smallest possible set of devices that can be distinguished by the IOMMU (e.g., all devices behind a PCIe-to-PCI bridge belong to the same group). Furthermore, each IOMMU group is assigned to an IOMMU domain.

Linux has a number of kernel boot parameters to change its default behavior in relation to IOMMUs. (SW-)IOMMUs can be disabled or enforced, the default domain can be set to pass-through such that devices bypass the IOMMU if not specified otherwise, and depending on the driver, the policy of IOTLB management can be changed as Linux defers IOTLB invalidation by default which increases performance but creates a window for invalid DMA accesses, i.e., devices may access some DMA-able memory that has been unmapped already as requests are still translated by the IOTLB. For the Intel IOMMU, IOTLB invalidation can be set to strict mode to enforce immediate flushing of

the IOTLB. The AMD IOMMU driver provides a similar option called `fullflush`. If no deviating behavior is specified, Linux flushes the IOTLBs every 10 ms or every 256 batched invalidation requests.

Device drivers do not configure IOMMUs directly. Instead, they change the IOMMU configuration through Linux's DMA-API by allocating, mapping and unmapping DMA memory, enforcing action by the respective IOMMU driver(s) depending on the operation and settings (e.g., immediate flushing of all IOTLBs).

DMA-mapping functions of the kernel return physical addresses as DMA address when the PCIe device uses no IOMMU and IOVAs with IOMMU. In both cases, memory is allocated using one of the kernel memory allocator functions. Only in the second case, however, an IOVA from the device IOMMU domain is allocated and the mapping from IOVA to physical address of the allocated memory is created through the IOMMU driver.

For each IOMMU domain, a base address and the upper boundary of device-addressable IOVAs are assigned on initialization. Allocated IOVAs ranges are stored in a red-black tree which on allocation is searched for a fitting address range bottom-up, i.e., from the highest IOVA range to the lowest. Since the node of the last allocated IOVA range is cached, most allocations happen in constant time [28]. To inhibit race conditions, access to the global IOVA red-black tree is fenced by a spinlock. To speed up IOVA allocation on multi-core systems, i.e., avoid locking and unlocking the spinlock, magazines are used, i.e., per core caches for previously deallocated IOVAs.

For controlled device access in user space, e.g., by user space network drivers, Linux offers a framework called Virtual Function I/O (VFIO). The framework consists of a device driver and various `ioctl`s to interact with IOMMU-controlled PCIe devices.

Devices to be used through VFIO have to be bound to the VFIO device driver. If a device's group contains multiple devices, all have to be bound to the driver or must be unbound from other host drivers (which will make the group available except the unbound devices). Device groups successfully bound to the VFIO driver appear as files in Linux's device filesystem at `/dev/vfio/{group-number}`. By chowning a device's group file to the current user, unprivileged access to the PCIe devices of the group is granted in a safe manner.

In VFIO, every IOMMU group in use belongs to a container. Depending on the IOMMU driver's capabilities, one or multiple groups may be encapsulated into a container, and containers may contain groups of different IOMMU domains.

To access a VFIO-bound device in user space, multiple steps have to be carried out by an application. The steps follow this scheme: First, a VFIO container is created and

initialized for the device to be used or an existing container is re-used. Second, the device's group file is opened by the user space application and the group is added to the VFIO container. Third, a file descriptor to the device is derived through a VFIO `ioctl` call on the group file. Fourth, the device file descriptor is used to map the device's registers into virtual memory. Finally, memory for DMA operations may be allocated in user space and IOMMU mappings to that memory may be created through the VFIO-API, returning IOVAs that can be passed to the device.

Internally, VFIO stores its IOVA mappings in a red-black tree. Multiple mappings may be coalesced to reduce tracking overhead. When a new mapping is requested through the VFIO-API, the red-black tree is checked for overlapping mappings, validity of the requested IOVA range is verified (e.g., supported address width by affected IOMMUs vs. provided IOVA address width), the mapping is inserted into the tree, pages are pinned for DMA access and the mapping is created in the IOMMU tables via the kernel-internal IOMMU-API.

2.7 Single-Root Input/Output Virtualization

Single root input/output virtualization (SR-IOV) is a mechanism that allows PCIe devices to appear as multiple devices and thus be shared with multiple virtual machines. Devices that support SR-IOV report this ability to the host via their PCI configuration space which includes registers for SR-IOV Extended Capability.

When SR-IOV is enabled for a device, the device can be split into a physical function (PF) and multiple VFs. The PF is the standard PCIe function of the device which has full control over the device, is used to reset and initialize it and to enable and disable VFs. Hence, access to the PF is usually restricted to OS or virtual machine manager (VMM). VFs on the other hand only support a subset of device operations, primarily to manage packet reception and transmission of the function, and can be passed to virtual machines in a safe manner.

For VFs, separate drivers are used as VFs use different device registers and have to communicate with the PF driver for certain operations like reset of the VF. A PF driver on the other hand must be able to handle these requests and is hence slightly more complex than the same driver without SR-IOV support.

With `ixgbe` for example, VFs of SR-IOV-capable devices can be created when loading the PF kernel driver: `modprobe ixgbe max_vfs=2`, which results in the PF driver creating two VFs for every `ixgbe`-bound device in the system, or by writing the required number of VFs into a device `sysfs` file called `sriov_numvfs`.

When VFs are created by the PF driver, every VF is assigned a PCIe BDF triple such that the VF can be identified on the PCIe bus and adressed directly. Although the function part of the BDF is only 3 bit, i.e., eight different functions, more VFs can be created by using different device numbers for the VFs.

Usually, the PF also assigns medium access control (MAC) addresses to the VFs and configures the NIC such that each VF receives the packets destined to its MAC address. To prevent MAC address spoofing, outgoing packets must be sent from the VF's MAC address and are dropped by the NIC in case VFs do not comply. The MAC anti-spoofing mechanism can be disabled by the PF driver.

While not required, SR-IOV can be used in conjunction with IOMMUs to pass VFs safely to virtual machines. When devices are passed-through without IOMMUs, virtual machines have unlimited access to the machine's main memory through the virtual function (see Section 2.4).

Chapter 3

Related Work

Publications on IOMMUs can be split into four categories: (1) basic documentation and evaluation of IOMMU utilization in Linux and Xen, (2) performance analyses of IOMMUs, (3) IOMMU emulation and usage in the context of virtualization, and (4) OS vulnerabilities by incorrect use of IOMMUs.

Early papers covered Sun Microsystem's IOMMU on SPARC stations of the Sun-4 architecture which was launched in 1987 [32, 25]. Miller and de Icaza document their efforts to port Linux to the SPARC platform, describing implementation hurdles and differences between various versions of the Sun-4 architecture [32]. They note that on larger machines the Sun IOMMU forms a separate MMU on the chipset, whereas on smaller machines it is included in the processor.

In 2006, research shifted to the previously released IOMMUs from Intel, AMD and IBM that marked a paradigm change: While former IOMMUs had been primarily used for address translation to enable DMA devices to access all physical memory of a host, these new IOMMUs were also capable of device isolation, i.e. restricting memory access of devices to device-specific memory regions [6]. Performance evaluations of these IOMMUs focused on CPU load: Ben-Yehuda et al. claim that IBM's Calgary/DART IOMMUs do not affect I/O throughput, however, CPU utilization increases by up to 60% in some scenarios [5]. They identify IOVA (de-)allocation as the primary factor of high CPU load and thus propose batching memory map and unmap calls or eliminating them wherever possible.

Later scientific work revealed more bottlenecks in hardware and software. Amit et al. focus on the IOMMU IOTLB [2]. They note that the TLB of their Intel Xeon X5570 IOMMU gets thrashed when more than 16 pages are used, increasing CPU execution

time by up to 47%. To evaluate different replacement strategies for IOTLB entries, they implement a virtual IOMMU for the KVM hypervisor which lets them trace map/unmap operations and analyse I/O memory access patterns. Based on their results, they propose changes in software and hardware that can reduce IOTLB misses by over 60%.

In a follow-up paper [3], Amit et al. use their virtual IOMMU to investigate different mapping strategies and performance/security implications of delayed IOTLB invalidations. They conclude that delaying IOMMU unmappings for a few milliseconds can lead to a significantly higher througput if DMA buffers are reused often (and are consequently not unmapped and invalidated in the IOTLB).

For devices with ring buffers, Malka et al. propose a new IOMMU design with drastically reduced overhead for IOVA (de-)allocation and a one-entry IOTLB per memory ring [29] to keep up with the increasing speeds of I/O peripherals.

Peleg et al. take a look at concurrent IOVA assignment and IOTLB management for scalable use of IOMMUs [37]. They propose three designs for IOVA assignment and a new IOTLB invalidation scheme.

Other researchers focus on security. Markettos et al. investigate the handling of IOMMUs by operating systems [30]. They build their own FPGA platform to attack systems as malicious Thunderbolt peripherals, and discover eight major vulnerabilities in Windows, Linux and macOS.

Chapter 4

Implementations

To analyse performance, safety and security implications of IOMMUs, we use `ixy.rs` [13], a high-speed user space network driver written in Rust for Intel 10G NICs of the `ixgbe` family, and adapt it to our needs. Changes to the existing implementation and new code are described in the following sections. All architectural and OS-related remarks refer to Linux on x86-64, the target platform of `ixy.rs`. The code of our implementations is available online [12, 11].

4.1 ixy.rs

Stefan Huber added basic IOMMU support to `ixy.rs` in 2019 through Linux's VFIO API [18]. We complete the implementation by adding support for the following two use cases: accessing multiple devices associated to the same IOMMU group and using `ixy.rs` in conjunction with legacy IOMMUs not supporting 48-bit wide IOVAs for second-level address translation (i.e., the translation mode used by VFIO).

For the former, only minor changes in the code are necessary. In the original implementation, a device's VFIO group file is opened on device setup and closed after the VFIO device file descriptor was derived from the group. However, a device group file cannot be re-opened if it is used through a derived device file descriptor, even if the descriptor of the group file was properly closed. Re-opening the group file in that case returns OS error 16, i.e., "device or resource busy". We mitigate this problem by keeping VFIO group files opened and storing the file descriptors in a `HashMap<i32, RawFd>` that maps IOMMU groups to group file descriptors. When a new device is initialized, the `HashMap` is checked for the VFIO group. In case there is no key-value pair for the group, the group

file is opened and group number and descriptor are added to the `HashMap`. Otherwise, the group file descriptor from the `HashMap` is used.

To use `ixy.rs` with IOMMUs with a maximum second-level translation IOVA widths smaller than the host's virtual address widths (e.g., only 39 instead of 48 bit), we reduce the IOVA widths in `ixy.rs`. Currently, `ixy.rs` uses a 1:1 mapping of process virtual addresses to IOVAs when setting up the DMARs. Hence, MMU and IOMMU map the same (I/O) virtual addresses to the same physical addresses and there is no need for `ixy.rs` to track which IOVA is used for which DMA buffer. We keep this mapping for simplicity but enforce all memory to be used by the NIC to be mapped into the lower 32 bit of the process address space. To our knowledge, there are no IOMMUs for the x86 architecture that do not support addresses up to 39 bit.

On a side note: a 1:1 mapping of process virtual addresses to IOVAs is one of the use cases mentioned in Intel's IOMMU specification as Shared Virtual Memory (SVM) [22, p. 21]. Unfortunately, we cannot make use of SVM since neither the `ixgbe` NICs nor Linux's VFIO framework support the required PCIe feature (PASID). In general, PASID seems to be a rather rare feature: Rothwell states in his dissertation that he has not seen it used in practice [38, p. 31].

Mapping memory into the 32-bit address space of a process is not rocket science. `mmap` can be forced through the `MAP_32BIT` flag to create mappings in the first 4 GiB of the process address space (indeed, only 2 GiB are used, which is why some authors call it the `MAP_31BIT` flag). However, mapping memory gets more sophisticated when huge pages are used (as they are in `ixy.rs`) due to alignment requirements: huge pages have to be huge-page-aligned, i.e., at 2 MiB or 1 GiB boundaries. Due to the alignment requirements, calling `mmap` with `MAP_32BIT` and `MAP_HUGETLB` fails.

To obey the alignment, we use `mmap` twice: to determine an appropriate address range and to map the huge pages into the address range. Listing 4.1 illustrates the algorithm. First, a mapping of needed size plus one huge page size is created (where needed size is equivalent to one or multiple huge pages). We know that any address range of size s contains a s-aligned address, i.e., for any address r returned by `mmap` we know that $\exists a \in \{r, r+1, ..., r+s-1\} : a \equiv 0 \mod s$. Thus we use the additionally mapped huge page to find the huge-page-aligned address a. After that, excess memory is freed, i.e., memory between the `mmap`- returned address and the huge-page-aligned address and any remaining bytes of the additionaly mapped huge page at the end of the mapping. Finally, huge pages of the needed size are mapped to the huge-page-aligned address by a second call to `mmap`, passing the aligned address as first argument to `mmap` and the flag `MAP_FIXED` to force `mmap` to map the huge pages at the specified address.

```
let addr = unsafe {
    libc::mmap(
        ptr::null_mut(),
        size + HUGE_PAGE_SIZE,
        libc::PROT_READ | libc::PROT_WRITE,
        libc::MAP_PRIVATE | libc::MAP_ANONYMOUS | libc::MAP_32BIT,
        -1,
        0,
    )
};

// calculate the huge page size aligned address by rounding up
let aligned_addr = ((addr as isize + HUGE_PAGE_SIZE as isize - 1)
    & -(HUGE_PAGE_SIZE as isize))
    as *mut libc::c_void;

let free_chunk_size = aligned_addr as usize - addr as usize;

// free unneeded pages (i.e. all chunks of the additionally mapped huge page)
unsafe {
    libc::munmap(addr, free_chunk_size);
    libc::munmap(aligned_addr.add(size), HUGE_PAGE_SIZE - free_chunk_size);
}

// finally map huge pages at the huge page size aligned 32 bit address
unsafe {
    libc::mmap(
        aligned_addr as *mut libc::c_void,
        size,
        libc::PROT_READ | libc::PROT_WRITE,
        libc::MAP_SHARED
            | libc::MAP_ANONYMOUS
            | libc::MAP_HUGETLB
            | MAP_HUGE_2MB
            | libc::MAP_FIXED,
        -1,
        0,
    )
}
```

Listing 4.1: Mapping huge pages into the first 2 GiB of the process address space.

Besides these newly implemented features in `ixy.rs`, some minor changes were made to the source code. We fixed a bug in the memory module where virtual addresses where used instead of I/O virtual addresses, a wrong bitmask was discovered and corrected in the original ixy driver [16] that propagated into all re-implementations [15] of the driver, and some comments were updated, typos fixed and overall code cleanliness improved (removal of superfluous casts, parentheses, etc.).

4.2 IXGBEVF

To analyse the effects of IOMMUs in context of SR-IOV, we implement `ixgbevf` for `ixy.rs`, a variant of the `ixgbe` driver for SR-IOV VFs. We base our implementation

on the `ixgbevf` code in the Linux kernel and DPDK, and the Intel 82599 SR-IOV Companion Guide [21].

While `ixgbe` (PF driver) and `ixgbevf` (VF driver) share large parts of their code base, device setup and communication between driver and NIC differ, and there are some tasks on VFs (like reset) that have to be executed cooperatively by both drivers. Besides these differences in the drivers, there are also differences in the devices: VFs support only up to 8 RX/TX queues in contrast to 64 of the PF, and the PCI configuration space of VFs returns `0xffff` for vendor and device ID, making it necessary to grab the values from files in the `sysfs` or from the PFs' PCI configuration spaces to determine the correct driver to be used.

Reset and initialization of the PF, i.e., the device itself, and creation of VFs is initiated by the PF driver. The PF driver brings the NIC into an operational state and creates VFs when requested to do so, e.g., by the OS.

Reset and initialization of the VF is initiated by the VF driver. To reset the VF, communication with the PF driver is necessary. In general, VF driver communication happens in two ways: Directly with the NIC via memory-mapped I/O and by passing messages to the PF driver via a mailbox system. For direct communication with the NIC, only a part of the device configuration space can be accessed by the VF. The registers in this configuration space allow the VF driver to set up the descriptor rings, enable or disable interrupts, send and receive packets, etc. For operations that have global impact like configuring VLAN filters or resetting the VF, the VF driver has to communicate with the PF driver and request the needed operations. This is done via a mailbox system that is implemented in hardware in the NIC.

Messages used for mailbox communication are hardware-independent and can be freely chosen by driver implementations. However, to be able to use our driver in conjunction with the kernel `ixgbe` driver as PF driver, we have to use the same messages for the same operations. Hence, we implement the mailbox in the way the Linux kernel driver and DPDK do.

Listing 4.2 shows how communication with the NIC mailbox is implemented in `ixgbevf`. `read_msg_from_mbx` is used to read a message from the mailbox buffer. It is expected to be called once a new message for the VF arrives (which can be determined through the `IXGBE_VFMAILBOX` register). The method takes a slice `msg` of unknown size to store the message from the mailbox. Before the message is copied to the slice, the mailbox is locked to prevent race conditions between VF and PF. Subsequently, the message is copied and receipt of the message acknowledged (releasing the mailbox at the same time).

```
fn read_msg_from_mbx(&self, msg: &mut [u32]) -> Result<(), Box<dyn Error>> {
    let len = min(msg.len(), self.mbx.borrow().size as usize);

    self.obtain_mbx_lock()?;

    for (idx, el) in msg[0..len].iter_mut().enumerate() {
        *el = self.get_reg32_array(IXGBE_VFMBMEM, idx as u32);
    }

    self.set_reg32(IXGBE_VFMAILBOX, IXGBE_VFMAILBOX_ACK);

    Ok(())
}
```

Listing 4.2: Reading a message from the VF mailbox.

VF initialization of `ixgbevf` resembles to a large extent PF initialization of `ixgbe`: Once the VF's registers are mapped into memory and DMA bus master is enabled for the VF, interrupts are disabled and the VF is reset by prompting the PF driver for function reset. When the PF driver has acknowledged function reset, the MAC address of the VF is derived from the PF through mailbox communication or generated locally. Subsequently, RX and TX descriptor rings are set up and initialization of the VF is complete.

Unlike in `ixgbe`, there is no link setup since the PF driver is responsible for the link. Indeed, VFs can only be used once the device's link is up. If the link is down, VF reset stalls. If the link goes down while VFs are used, the VFs become inoperable and have to be reset once the PF's link is up again.

Deriving the MAC address and using it as source address for outgoing packets is crucial for `ixgbevf` as the kernel driver enables MAC anti-spoofing by default, i.e., all TX packets are checked for deviating MAC addresses by the NIC and get discarded if a deviating address was detected.

4.3 IOMMU-LEAKS

For precise time measurements on NIC operations and to analyse effects of the IOTLB, we implement some additional functionality in `ixy.rs`.

To be independent of the physical link, we add methods to enable loopback operations on the NIC. In loopback mode, packets are transmitted from the NIC's TX queues via its internal 10 Gigabit Media Independent Interface (XGMII) to the RX queues without leaving the device.

We also add methods to dynamically enable and disable queues. This is advantageous to approximate how much time is needed for DMA transfers. By disabling all RX queues,

TX transmit times can be broken down to a DMA access to the TX queue page and another DMA access to the packet buffer.

To measure CPU cycles for NIC operations, we use the x86 `rdtsc` instruction. We implement a rdtsc function as shown in Listing 4.3. We use intrinsic functions to avoid inline assembler. Since intrinsic functions are inherently unsafe, we need an `unsafe` block. Note our use of memory fences to serialize instruction execution. `lfence`s deter the compiler from reordering code and the CPU from speculatively executing subsequent instructions, namely the `rdtsc` itself and following instructions. Unlike suggested by a Intel in a benchmarking guide for the IA-32 and IA-64 architecture [35], we use `lfence` instead of `cpuid` as it provides the same serialization guarantees and keeps the variance small.

```
#[cfg(all(any(target_arch = "x86", target_arch = "x86_64")))]
#[inline(always)]
fn rdtsc() -> u64 {
    #[cfg(target_arch = "x86")]
    use core::arch::x86;
    #[cfg(target_arch = "x86_64")]
    use core::arch::x86_64 as x86;

    unsafe {
        x86::_mm_lfence();
        let rdtsc = x86::_rdtsc();
        x86::_mm_lfence();

        rdtsc
    }
}
```

Listing 4.3: `rdtsc()` function to count CPU cycles.

To be able to evaluate measured CPU cycles, we implement a logger that logs values to a file-backed huge page and provides statistics about the measured values like mean, variance and sample variance.

For IOTLB-related research, we implement a brute-force memory allocator that allocates physically and virtually contiguous 4 KiB pages. By using normal sized pages, locality of packet buffers can be maintained while the amount of pages to be translated by the IOMMU respectively the IOTLB can be controlled variably.

The memory allocator works as follows: First, two memory regions are allocated via `mmap`, a huge region – which is our page pool – containing 1,024 pages that are checked for contiguity and a target memory area. Next, virtual and physical address of every page in the page pool are inserted into some data structure, e.g., a vector, and sorted by ascending physical addresses. Subsequently, the sorted pages are remapped to the target memory area using `mremap` and checked for contiguity. Since all pages have been remapped in physically contiguous order, physically adjacent pages are now also

adjacent in virtual address space. Thus, it suffices to iterate through the target memory area and check whether there is a block of physically subsequent pages big enough to satisfy the requested amount of contiguous memory.

To control memory placement of RX/TX queues and packet buffers, we add methods to `ixy.rs` that reinitialize the aforementioned data structures at variable memory addresses, e.g., memory allocated by our brute-force memory allocator. We also add methods for fine-grained control of TX descriptors and queues that allow us to repeatedly use the same descriptor(s) for transmit.

Chapter 5

Performance Analysis

To determine the performance impact of IOMMUs on throughput and latency, we conduct various measurements using the implementations described in Chapter 4, i.e., `ixy.rs`, `ixgbevf` and `iommu-leaks`.

5.1 Methodology

All measurements are performed on server pairs connected via two 10 Gbit/s links. The devices under test use Intel NICs of the `ixgbe` family that are compatible with our drivers, i.e., X520, X540 and X550. Each device under test is equipped with two of these cards since two separate NICs yield better performance than a dual-ported one (probably due to hardware limitations of the NIC or the PCIe bus). Table 5.1 lists the devices under test.

On the devices under test, we run the different implementations. For the measurements, two applications are used on top of the drivers, a traffic generator and a forwarder. The traffic generator allocates a memory pool and repeatedly sends out the same buffers,

CPU	**Year**	**Arch.**	**Memory**	**NIC**	**NUMA**
Intel Xeon E3-1230v2	2012	Ivy Bridge	16 GB	Intel X520-DA1 Intel X520-DA2	no
Intel Xeon E5-2620v3	2014	Haswell	32 GB	Intel X520-DA2 Intel X540-T2	no
AMD EPYC 7551P	2017	Naples	128 GB	Intel X550T Intel X550T	yes

Table 5.1: System configurations of servers used for performance analysis.

CPU	Clock	Cores	L3-Cache	PassMark ST	PassMark All
Intel Xeon E3-1230v2	3.3 GHz	4	8 MB	1,996	6,192
Intel Xeon E5-2620v3	2.4 GHz	6	15 MB	1,700	7,979
AMD EPYC 7551P	2.0 GHz	32	64 MB	1,611	25,933

Table 5.2: CPUs of devices under test and their PassMark scores for single thread (ST) and all cores.

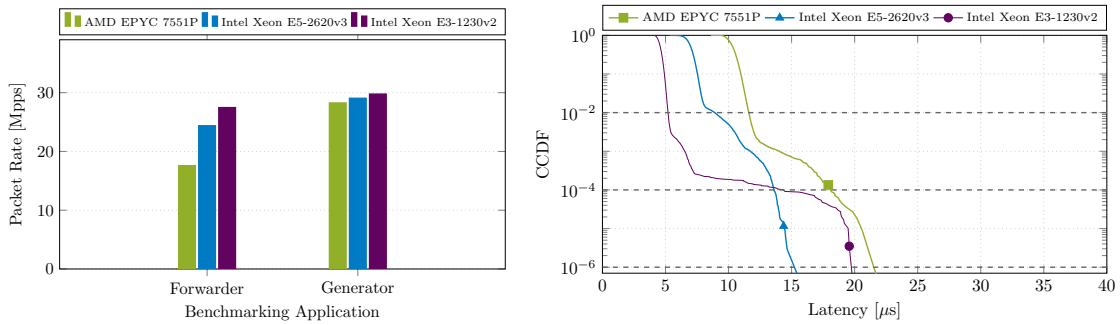
only updating the sequence numbers of the packets. The forwarder uses two devices, allocates a pool per device and forwards packets bidirectionally, i.e., packets received by the first device are sent out by the second and vice-versa. To simulate a somewhat realistic workload, one byte of each packet is increased, enforcing the CPU to load the byte into the CPU caches. On the second server, MoonGen [14] generates traffic with its `l2-load-latency.lua` script.

All benchmarks are conducted at native CPU frequency with dynamic overclocking disabled. CPU cores are pinned for the measurements. On NUMA architectures, i.e., the AMD server, locality is obeyed for CPU core pinning and memory placement. Table 5.2 contains details of the CPUs. Two important characteristics emerge from the table: clockspeed has a huge impact on single thread/core performance while the absolute number of cores is crucial for overall CPU performance.

Our benchmarking applications are executed on a single CPU core in a single thread as full support for multi-core systems is not implemented in the drivers yet. On most systems this is not a limitation as hardware limits can be hit with a single thread. It stands out from Table 5.2 that – although being the oldest of the three CPUs – the Intel Xeon E3 yields the highest per-thread performance while the AMD CPU performs worst. This observation is backed by our baseline measurements (Section 5.2).

To reduce architecture-dependent overhead and assimilate the results of our measurements, packet-prefetching is disabled in all implementations. If not specified otherwise, packets are processed in batches of 32, ring sizes of 512 are used for RX and TX descriptors, and all DMA-able memory is allocated on 2 MiB huge pages. When the RX packet rate exceeds the host's packet processing rate, i.e., the RX queue is full, packets are dropped by the NIC.

Throughput and latency results of the implementations are measured on the second server by MoonGen. Forwarded and generated packets contain 60 B of data, i.e., 84 B with checksums and interpacket gap which is the minimum Ethernet frame size, such that line rate at 14.88 Mpps can be achieved in all measurements.



(a) Throughput (b) Latency

Figure 5.1: Baseline throughput and latency of servers used for performance measurements. Latency of forwarder measured with a total packet rate of 10 Mpps.

For measurements including an IOMMU, pass-through mode is used for the IOMMU to reduce side effects from other I/O devices on the host. All measurements are conducted with Rust 1.50.0 on Debian Buster 10.6 with a Linux 4.19 kernel.

5.2 Baseline Measurements

To determine the baseline performance of our devices under test, we run the forwarder and the generator application in default configuration on all servers. Figure 5.1 shows the results. Since the generator application generates traffic on a single port only, we run two instances on two different devices for our measurements. Consequently, the bars on the left of Figure 5.1a show single core forwarding rate while the bars on the right show dual core packet generation rate. Since the two instances of the packet generator are executed independently of each other on two different CPU cores and devices, single core packet generation rate equals (approximately) the measured results divided by two.

From Figure 5.1a, it is particularly striking that the packet forwarder on the AMD EPYC CPU (packet rate: 17.6 Mpps) is about one third slower than on the Intel Xeon E5 CPU (packet rate: 24.4 Mpps) although clockspeeds differ only by one sixth.

We use CPU performance counters to profile the forwarder with both CPUs and identify two reasons for the poor performance of the AMD CPU: the AMD CPU executes about 1.75 instructions per cycle (IPC) while the Intel CPU runs about 2.1, and while 0.01-0.03% of all branches are mispredicted on the Intel CPU, the AMD CPU has a misprediction rate of 0.40%.

Although AMD EPYC's Naples architecture is based on Zen, we find reports on poor IPC counts and branch misses for the successor architecture of Zen – Zen 2 – which

seem to support our findings [27, 26]. Compared to Intel Skylake, the AMD Zen 2 CPU seems to take one to two additional cycles per branch misprediction.

Looking at Figure 5.1a, both Intel CPUs stand out, with the cheaper Xeon E3 CPU introducing more variance into latency while the more expensive Xeon E5 CPU forwards traffic rather smoothly.

5.3 Page Sizes

It is well known that page sizes impact system performance. Page sizes are a trade-off: When larger pages are used, overhead for page management decreases, and page tables and the TLB have to keep less entries. At the same time, fragmentation and page swapping increases as even tiny amounts of data have to be stored on whole pages and less pages can be kept in main memory at a time. Smaller pages on the other hand increase page tables and put more pressure on the TLB since more pages have to be accessed for the same amount of data. This can lead to TLB thrashing: address translations are continuously replaced by new translations in the TLB, reducing effectiveness of the cache. In extreme cases, addresses are re-translated for every page access, i.e., the TLB is practically no longer in use.

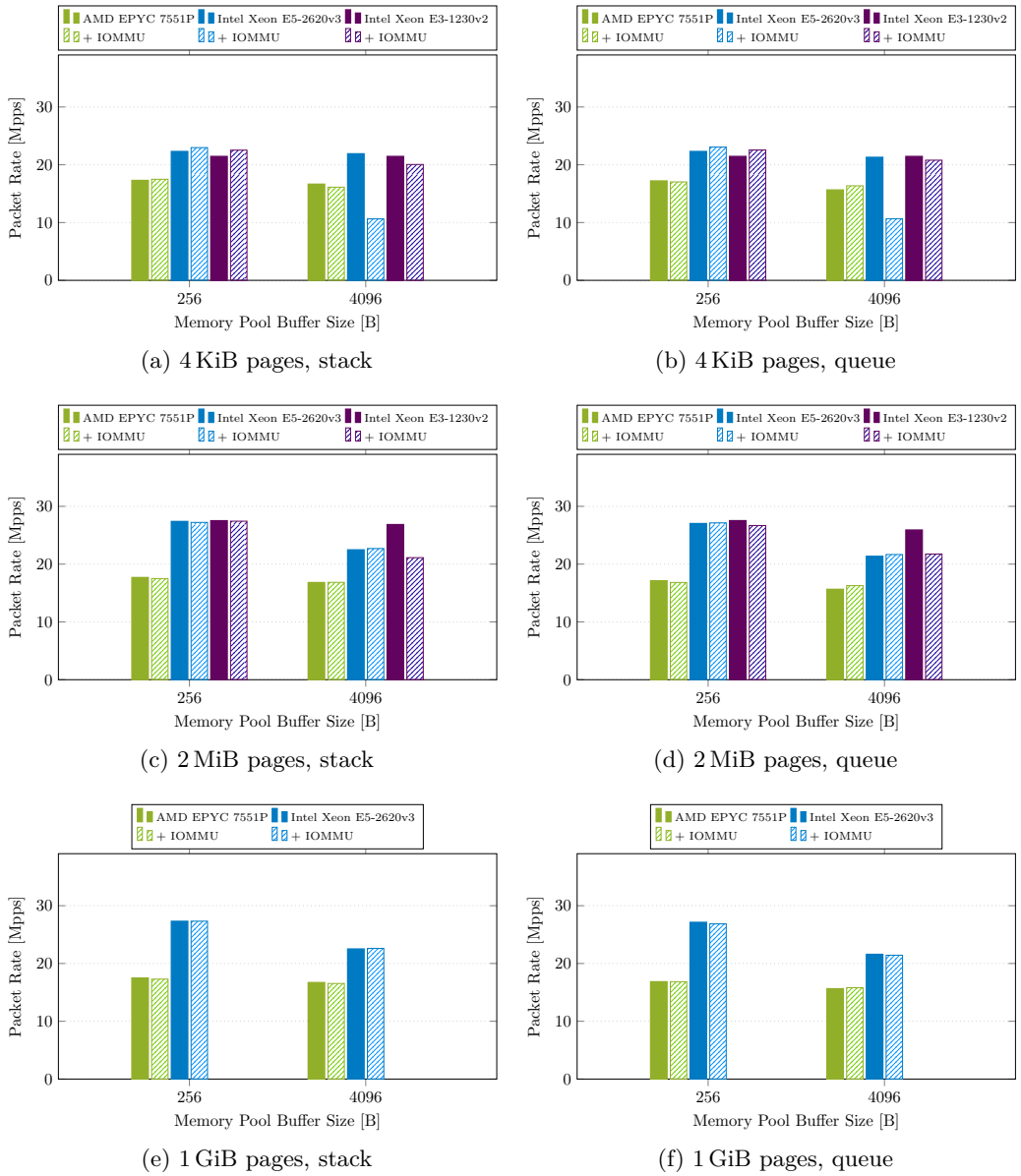
Available page sizes on a system depend on processor architecture and OS. Linux on x86 uses 4 KiB pages by default. With huge pages, sizes of 2 MiB or 1 GiB can be used. These page sizes are also supported by Intel's and AMD's IOMMUs.

To measure effects of different page sizes and number of pages, we run the forwarder application with 2 MiB and 1 GiB huge pages as well as our brute-force memory allocator to allocate memory on 4 KiB pages. Figure 5.2 visualizes packet rates for different page sizes, memory pool buffer sizes and data structures for the memory pool. For all measurements, RX and TX rings have a queue size of 256 descriptors. Figure 5.2a, Figure 5.2c and Figure 5.2e use the default implementation of the driver, i.e., a stack data structure for the free buffers in the memory pool. There are no results for the Intel Xeon E3 CPU and 1 GiB pages since these page are not supported by the CPU.

We run our measurements with memory pool buffer sizes of 256 B and 4096 B. With 4 KiB pages, different buffer sizes force the driver to allocate varying numbers of DMA-able pages.

The exact number of allocated pages is determined by page size, buffer size and number of descriptors. In case of huge pages, three huge pages are used per device as all data structures fit onto single pages: one page is allocated for the RX descriptor ring, one for the TX descriptor ring and one for the packet memory pool. In case of 4 KiB pages,

1 page is used for each descriptor ring as each descriptor is 16 B and thus 256 descriptors fit onto a single page. The number of pages allocated for the memory pool depends on the buffer size. Since `ixy.rs` allocates the memory pools by default with twice as many buffers as RX/TX descriptors, the pool contains 512 packet buffers. With a buffer size of 256 B, that totals to 128 KiB, i.e., 32 pages. With a buffer size of 4,096, 512 pages



(a) 4 KiB pages, stack

(b) 4 KiB pages, queue

(c) 2 MiB pages, stack

(d) 2 MiB pages, queue

(e) 1 GiB pages, stack

(f) 1 GiB pages, queue

Figure 5.2: Throughput of forwarder with different page sizes, memory pool buffer sizes and data structures for the memory pool.

are allocated. In summary, 34 pages are allocated per device with 256 B memory pool buffers and 514 pages are allocated with 4096 B buffers.

Various conclusions can be drawn from Figure 5.2a, Figure 5.2c and Figure 5.2e. On the AMD server, neither page size nor buffer size seem to have a significant effect on throughput. The biggest performance drop happens with 4 KiB pages: Without IOMMU, packet rate between 256 B memory pool buffers, i.e., 68 pages in total, and 4096 B memory pool buffers, i.e., 1,028 pages, decreases by 0.7 Mpps. With IOMMU, packet rate decreases twice as much, i.e., by 1.4 Mpps.

On the Intel servers, results are inconsistent. The Xeon E5 CPU exhibits quite constant behavior with huge pages across all configurations: packet rates drop with increasing buffer sizes, possibly due to decreased data locality. Using 4 KiB pages without IOMMU, packet rate is almost constant while with IOMMU throughput goes down by more than 50% when using greater buffers, i.e., allocating more pages. We inspect this anomaly closer in Figure 5.3.

Like the AMD CPU, the Intel Xeon E3 delivers constant throughput rates in all non-IOMMU configurations independent of page size and buffer size. Compared to the Xeon E5 CPU, IOMMU performance seems abnormal for 4096 B memory pool buffers with 4 KiB and 2 MiB pages. For 4 KiB pages, packet rate is unexpectedly high while performance drops significantly for 2 MiB pages. The reasons for these unusual performance characteristics remain in the dark. Notably, both Intel CPUs profit from huge pages unlike the AMD CPU.

In summary, our measurements imply that larger memory pool buffers and thus more pages have a negative effect on packet throughput. In case of the IOMMU, performance can degrade by more than 50% on some CPUs. To back up our results, we repeat the measurements and replace the free stack of the memory pool by a queue. Our reasoning for changing the data structure is that with a stack, the number of used pages by driver and device is not necessarily equal to the number of allocated pages as there are probably some buffers at the bottom of the stack that will never be used. With the stack, we know that the minimum number of used pages is the number of pages used for buffers in the RX descriptor ring + the number of pages used for buffers in the TX descriptor ring + the number of pages used for received and still-in-use packets.

In case of the 34 pages configuration, the minimum number of used pages is 2 pages for the RX/TX descriptor rings, 16 pages for the buffers in the RX descriptor ring, 2 pages for a batch of 32 received packets, and 2 pages for a batch of 32 packets in the transmit queue (as the transmit queue is cleaned every 32 packets), i.e., 22 pages per device or 44 pages in total. In case of the 514 pages configuration, we conclude with the same

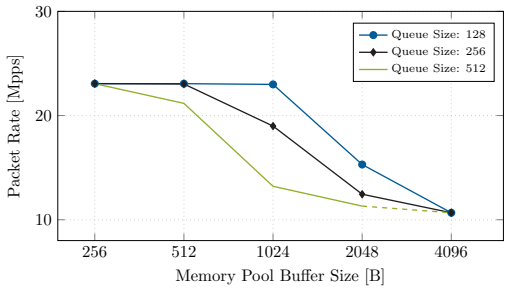


Figure 5.3: Throughput of forwarder on Intel Xeon E5-2620v3 with 4 KiB pages, IOMMU enabled, and using a queue for the memory pool.

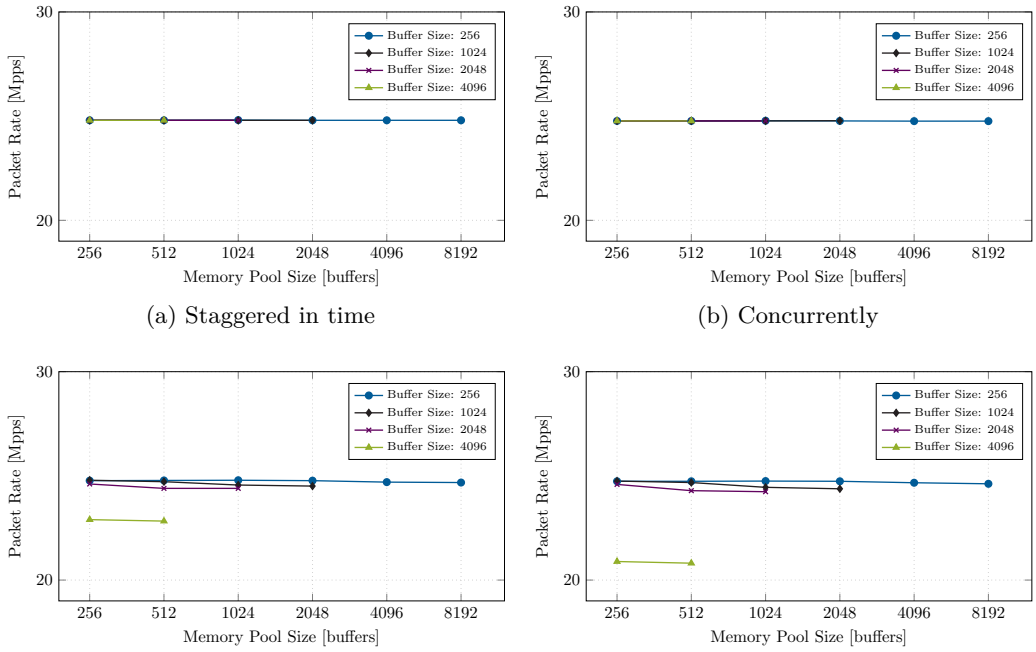
reasoning that a minimum of 322 pages per device or 644 pages in total is used. We see from our considerations that about two-thirds of the allocated pages have to be used by driver and device.

By implementing the free stack of the memory pool as a queue, we enforce that all allocated pages are used. Figure 5.2b, Figure 5.2d and Figure 5.2f show the results of our measurements using a queue for the free stack of the memory pool. Notably, there are only minor differences in performance between the two data structures. It seems worth mentioning that, in combination with the IOMMUs, there are four configurations in which throughput with the queue is even slightly higher than with the stack.

As noted before, Figure 5.2a and Figure 5.2b depict a drastical loss of performance with 4 KiB pages and larger memory pool buffer sizes on the Xeon E5 CPU when using the IOMMU. To determine whether this performance degradation is caused by the number of used pages, we run the forwarder application with different queue and memory pool buffer sizes on the Xeon E5 CPU, using 4 KiB pages, IOMMU and a queue as data structure for the memory pool. Figure 5.3 shows the results of our measurements.

The stepwise decrease in packet rate for different queue and memory pool buffer sizes seems to imply that the number of used pages is indeed the reason for loss of throughput. With 2 + 64 pages per device, i.e., a queue size of 128 and 1024 B buffers, a queue size of 256 and 512 B buffers or a queue size of 512 and 256 B buffers, throughput is at its maximum. With 2 + 128 pages, packet rate decreases between 1.9 and 7.7 Mpps. With 2 + 512 pages, throughput is at its worst with 10.7 Mpps.

We repeat our measurements with the packet generator application on the Xeon E5 CPU. Figure 5.4 shows the results of our measurements. We use a queue data structure for the memory pool and vary memory pool buffer and memory pool size to run the generator with different numbers of 4 KiB pages, ranging from 16 pages (256 buffers with 256 B each) to 512 pages (8,192 buffers with 256 B each, or 2,048 buffers with 1024 B each, etc.).



(a) Staggered in time

(b) Concurrently

(c) Staggered in time with IOMMU

(d) Concurrently with IOMMU

Figure 5.4: Packet rate of two generator instances on Intel Xeon E5-2620v3 with 4 KiB pages, different memory pool and buffer sizes.

We run two instances of the packet generator, each instance generating traffic on one device. We set the queue size of RX and TX descriptor ring to 256 entries. Figure 5.4a and Figure 5.4c show the results when running the generator instances staggered in time, Figure 5.4b and Figure 5.4d when running them concurrently.

To our surprise, there seems to be no correlation between performance degradation and the number of used pages. Packet rate in the non-IOMMU measurements is absolutely constant, with a loss of about 40k packets when running both generators concurrently, while with IOMMU, packet rate decreases slightly with larger memory pool sizes, and significantly with a buffer size of 4096 B. When running the two generators concurrently with 4096 B buffers, packet rate goes down by 3.7 Mpps compared to 256 B buffers using the same number of pages.

To verify whether packet rate in Figure 5.3 drops due to only one IOMMU domain being used for both devices, we modify the forwarder of Figure 5.3 such that every device belongs to another domain, and, since the devices can no longer access each other's DMA memory, mirror packets received by the devices back onto the link. We check packet rates in bullet point manner and note that there is almost no difference in performance (~0.1 Mpps) when using one or two domains for the devices. We also

modify the packet generator Figure 5.4 to generate traffic on two devices, i.e., using only one domain for both devices. Again, there is no noticeable performance difference between one and two domains being used.

Our final guess as to why packet rate drops with the forwarder but not the generator is that while both allocate 512 pages per device, the forwarder uses both memory pools by each device: packets received by the first device end up in a memory pool of the first device and are sent out from that pool by the second device and vice versa. Thus, each device accesses more than 1,024 pages, twice as many than with the generator. To confirm our suspicion, we modify the generator such that every device sends out packets in an alternating manner from two pools. Again, there is no noticeable drop of performance when increasing the number of pages.

In summary, we note that on the AMD EPYC CPU, there are no quantifiable effects of the IOMMU on performance. On the Intel CPUs, in some configurations packet throughput is impaired by the IOMMU by more than 50%. However, the reasons for this remain unclear: While our measurements in Figure 5.3 imply a correlation between packet rate and the numbers of pages in use, this is not confirmed by our packet generator.

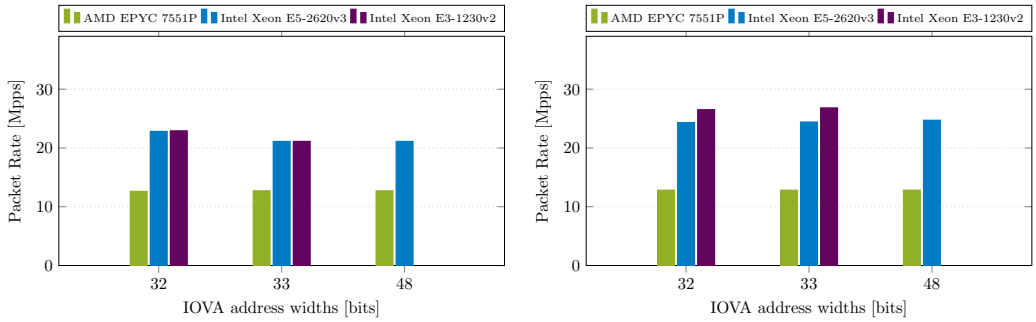
5.4 IOVA Address Widths

For IOVAs, any mappings can be chosen. Depending on page size and IOVA address widths, IOMMUs use different paging structures: To translate a 57-bit address to 4 KiB pages, an Intel IOMMU uses a 5-level paging structure, for 48-bit and 39-bit addresses, 4 and 3 levels are used respectively.

We run our packet forwarder application to determine the performance impact of different IOVA address widths. Figure 5.5 shows the results of our measurements. We run the forwarder on two ports of a single NIC (Figure 5.5a) and on two distinct NICs (Figure 5.5b). There are no values for the Intel Xeon E-3 CPU and 48-bit addresses as they exceed the maximum IOVA address widths of the Xeon's IOMMU which is 39 bit.

When using a single dual-ported NIC for the forwarder, packet rate indeed decreases by about 1.7 Mpps on both Intel CPUs with 48-bit addresses instead of 32-bit addresses. We repeat our measurements and note that the loss of performance occurs as soon as IOVA address widths surpass the 32-bit boundary.

When using two NICs for the forwarder, no significant change in performance can be observed. Throughput increases slightly on the Intel CPUs by 0.3 Mpps.



(a) Throughput using 1 NIC

(b) Throughput using 2 NICs

Figure 5.5: Throughput of forwarder with 32, 33 and 48 bit wide IOVAs.

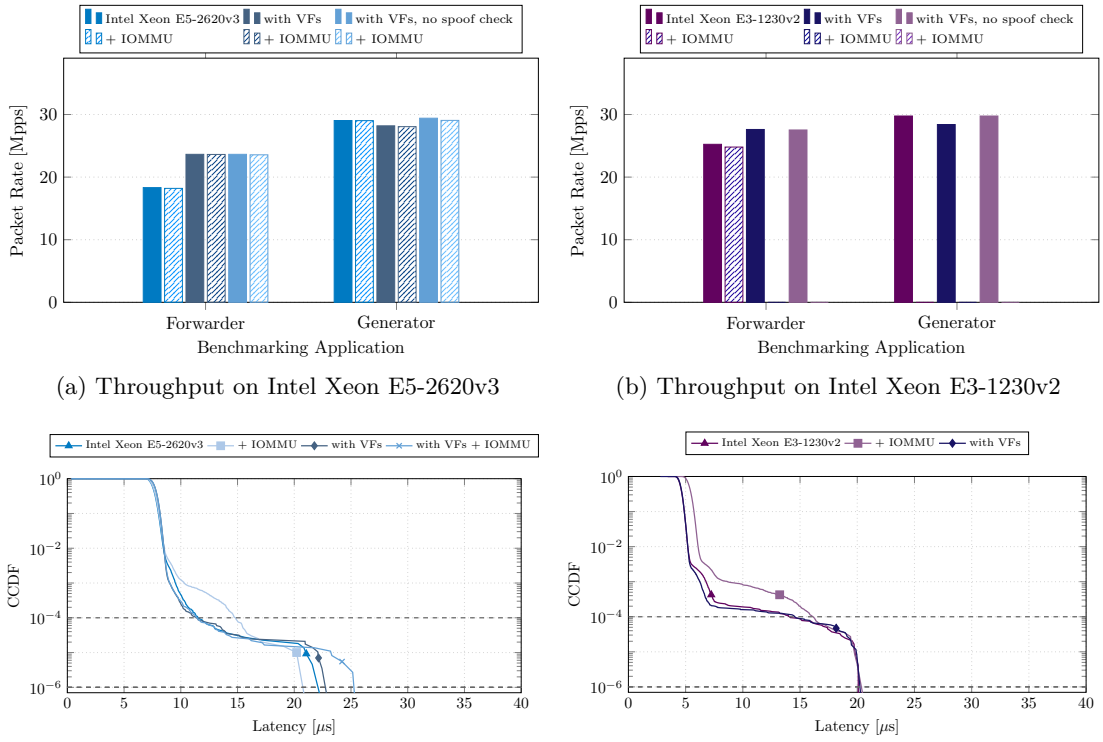
We take a closer look at Figure 5.5a with the different packet rates for different IOVA address widths. We note that 32 bit is not a boundary in the Intel IOMMU paging structure. Compared to Figure 5.5b, throughput of the forwarder is capped at a certain packet rate. We suspect that this is a bottleneck at the PCIe level. A sufficient explanation is given by the PCIe specification [36]: The header of PCIe memory read requests varies in size depending on the address widths, for 32-bit addresses it is 4 B smaller. Thus, more packets can be transmitted at the PCIe level.

We conclude from these results that changes in performance with different IOVA address widths are not related to the IOMMUs.

5.5 Virtualization

We take a look at IOMMU impact on virtualization, i.e., SR-IOV. For baseline measurements, we run the packet forwarder and packet generator application on top of two VFs, with one VF per device. We modify the applications slightly to work with SR-IOV: instead of incrementing one byte in the packets, the forwarder sets the source MAC address of each packet to its own address – otherwise, packets are dropped by the NIC due to its anti-spoofing mechanism.

Figure 5.6 shows the results of our measurements. The AMD EPYC CPU is missing as only one of the two PCIe ports on the board supports SR-IOV. Some values for the Intel Xeon E5 CPU are also missing due to IOMMU group restrictions: On our server, VFs and PFs of the NICs belong to the same IOMMU group, making it impossible to access them at the same time by multiple applications through VFIO, or through VFIO and the native driver for the PF.



(a) Throughput on Intel Xeon E5-2620v3

(b) Throughput on Intel Xeon E3-1230v2

(c) Latency on Intel Xeon E5-2620v3

(d) Latency on Intel Xeon E3-1230v2

Figure 5.6: Throughput and latency of modified forwarder with two devices, with and without VFs, with and without IOMMU. Latency of forwarder measured with a total packet rate of 10 Mpps.

As in our other measurements, two instances of the generator are run for the measurements, one per device. Notably, compared to the unmodified forwarder in Figure 5.1a, the modified forwarder is about 6 Mpps slower on the Intel Xeon E5 CPU, and about 2 Mpps slower on the Intel Xeon E3 CPU.

The packet rates show that the IOMMUs have almost no impact on performance, neither in the base configuration nor with VFs. Absurdly, packet rate with VFs is 5.3 Mpps greater on the Xeon E5 CPU than without VFs, and on the Xeon E3 CPU the forwarder with VFs reaches a new maximum of all measurements with 27.6 Mpps (0.15 Mpps faster compared to Figure 5.1).

When running the generator application, we note that throughput is capped at 14.20 Mpps per port. We disable the MAC anti-spoofing mechanism and reach line rate on the Xeon E3 CPU, i.e., 14.88 Mpps per port. Thus, we put the cost of the spoof check at 0.68 Mpps per port or 1.36 Mpps in total. We repeat this experiment on the AMD EPYC CPU and note that packet rate increases from 14.08 Mpps to 14.35 Mpps when disabling the

spoof check. To our surprise, on all hosts the PF driver of the Linux kernel reports some spoofed packets when switching the spoof check on or off on the fly.

We also measure the latency. On the Xeon E3, latency of forwarder, and forwarder with VFs resemble each other with, while latency of forwarder with IOMMU is on average 1 µs greater than without IOMMU. On the Xeon E5, latencies are also approximately the same, but tail latencies diverge. Interestingly, forwarder and IOMMU achieve the lowest tail latencies, followed by the forwarder as-is, forwarder with VFs and forwarder with VFs and IOMMU.

We conclude that IOMMUs do not have a significant effect on throughput and latency of VFs.

5.6 Summary

Regarding the AMD EPYC CPU, there are no measurable differences in performance with AMD's IOMMU. On Intel systems, IOMMUs have a negative impact on performance in certain configurations. We suspect a correlation between throughput and the amount of pages used by a device. To be more precise, we assume that the IOTLB gets thrashed the more pages are used.

This assumption is not unreasonable: The datasheet for the Intel 82599 NICs mentions changes in the PCIe transaction size to decrease the number of PCIe transactions and thus thrashing of the IOTLB [20, p. 34]. And Neugebauer et al. [34] report IOTLB thrashing on their Intel Xeon E5-2630v4 based system, and determine an IOTLB size of 64 entries for that particular IOMMU. Unfortunately, our measurements confirm the correlation between throughput and used pages only to some extent.

We were not able to identify any striking performance differences in relation to IOVA addresses or virtualization.

Chapter 6

Safety and Security

IOMMUs have been bypassed in numerous ways due to weaknesses in PCIe, boot-time issues and poor configuration of IOMMUs by OSes. We describe some attacks to give a short overview of vulnerabilities. In the second half of the chapter (Section 6.2), we take a closer look at the IOTLB of IOMMUs which we assume to be another weak point. We suspect that the IOTLB is susceptible to timing attacks.

6.1 Known Vulnerabilities

On the PCIe level, two flaws stand out. First, devices can impersonate other PCIe devices by spoofing their PCIe ID (i.e., the BDF). Since IOMMUs perform address translation and check access permissions based on the PCIe ID of PCIe packets, devices can access memory assigned to other devices by spoofing their ID. When IOMMUs are used in pass-through mode, i.e., only some devices use the IOMMU, devices may even bypass the IOMMU completely and access all memory of a host. Access in this context refers primarily to write access. For read access, routing configuration has to be updated. This is theoretically possible but rather unlikely in practice. Daubignard and Perez explain the attack vector in more detail [10].

Second, devices may implement a PCIe capability called Address Translation Services (ATS). With ATS, devices announce that they are capable of caching address translations and translate addresses of DMA requests on their own. Thus, ATS reduces pressure on the IOTLB of IOMMUs. Devices using this capability request translations from DMA remapping hardware, cache the translations and receive invalidation requests once address mappings are no longer valid. PCIe memory read and write requests of such devices indicate whether the specified address was translated by the device or still

needs to be translated (by the IOMMU). If ATS is enabled for a device, memory requests with translated addresses pass the remapping hardware unchecked. There are no restrictions on which addresses can be accessed by a device, malicious or faulty devices can masquerade any address as legitimately translated address.

Both flaws in the PCIe protocol can be mitigated (to a certain point) using a PCIe feature named Access Control Services (ACS). With ACS, ports of the root complex and switches can be configured to verify whether the PCIe ID used in a request belongs to a port's subset of bus numbers. Furthermore, memory requests with ATS-translated addresses can be blocked. These operations are known as source validation and translation blocking. However, ACS is an optional PCIe feature and might not or only partially be available on a system [36, p. 582].

In case ACS is not available, ATS can still be disabled in the IOMMU. Both Intel's and AMD's IOMMUs check memory requests with translated addresses for eligibility, i.e., whether ATS is enabled. Intel stores this information in the context entry of the domain, AMD in the device entry.

Other attack vectors exist at boot-time: On some architectures, IOMMUs could be hidden from the OS by rewriting the DMAR ACPI tables at boot-time [40]. Morgen et al. used another weakness and rewrote the tables of the IOMMU during initialization, setting the translation type of IOMMU domains to pass-through mode [33].

The flaws in how operating systems configure IOMMUs are even worse. In 2019, Markettos et al. examined the behavior of all major OSes and came to disillusioning conclusions [30]. Most versions of Microsoft Windows did not use IOMMUs at all and were therefore entirely unprotected against DMA attacks. Windows 10 Enterprise supported IOMMUs, however, they were only used to protect a secure container (VBS), not the root operating system. Unlike Windows, macOS and Linux enabled IOMMUs. However, DMA memory was shared between devices on macOS and both operating systems exposed OS data structures to the devices. Markettos et al. show 5 different ways on macOS and Linux to bypass the IOMMU, some so serious that a root shell could be obtained.

Following the revelations of Markettos et al., OS vendors released mitigations against the newly discovered vulnerabilities. Besides, Microsoft announced to support IOMMUs as of Windows 10 version 1803, and Linux disabled ATS entirely for external devices (e.g., Thunderbolt). However, internal hardware is still considered trustworthy. To reduce the surface area for attacks sustainably, Markettos et al. call for a fundamental change of the thread models of operating systems and drivers.

6.2 The IOTLB

We suspect that there exists yet another vulnerability in IOMMUs in relation to IOTLBs, which are used by IOMMUs to cache information about domains and address mappings to accelerate address lookup. To be more precise, we suspect that these IOTLBs – just like other caches – are vulnerable to timing attacks, i.e., multiple devices or functions share the same IOTLB and DMA access times depend on availability of translation information in the cache.

We implement an application on top of the methods described in Section 4.3 to run some timing measurements on DMA operations. Our application follows the idea of priming and probing: By carefully controlling the DMA accesses of the NIC, we try to bring the IOTLB into a known state. We run our measurements on the Intel Xeon E5 CPU as it is the only CPU that supports SR-IOV with IOMMU and two devices.

We assume that the IOTLB of this CPU has 64 entries. We allocate 64 physically and virtually contiguous 4 KiB pages, place the TX descriptor ring on the first page and use the other 63 pages for our packets. We use one port of the Intel X520-DA2 NIC for our application and configure the device to use our 64 pages. We disable the RX queue of the device to prevent DMA accesses on packet arrival.

In our first measurements, we try to determine the number of CPU cycles it takes for the NIC to fetch the TX descriptors from one page, transmit a batch of 63 packets from the 63 other pages, and set the descriptor-done bit in the last TX descriptor on the descriptor page. Thus, the NIC accesses 64 distinct but physically and virtually contiguous pages for every batch of to-be-transmitted packets.

We run the application with and without IOMMU. We warm up the caches before starting our measurements. For every run of the application, we measure the CPU cycles for 500k batches and mean and standard deviation of the CPU cycles. Figure 6.1 shows the results of five runs without IOMMU. The plots are cut off at 50,000 CPU cycles, all spikes that end up at 50,000 cycles go up to 400,000 CPU cycles.

Several conclusions can be drawn from the plots. First, the number of CPU cycles varies greatly in every run. Second, the number of CPU cycles between spikes is constant in a run and for all runs. Third, some spikes appear equidistantly across runs, e.g., with a time interval of 0.4, 0.6 or 2.0 s.

On average, we note 15,622 CPU cycles and a standard deviation of 623 cycles for the runs in Figure 6.1. With a batch size of 63 packets, we get 248 cycles per packet.

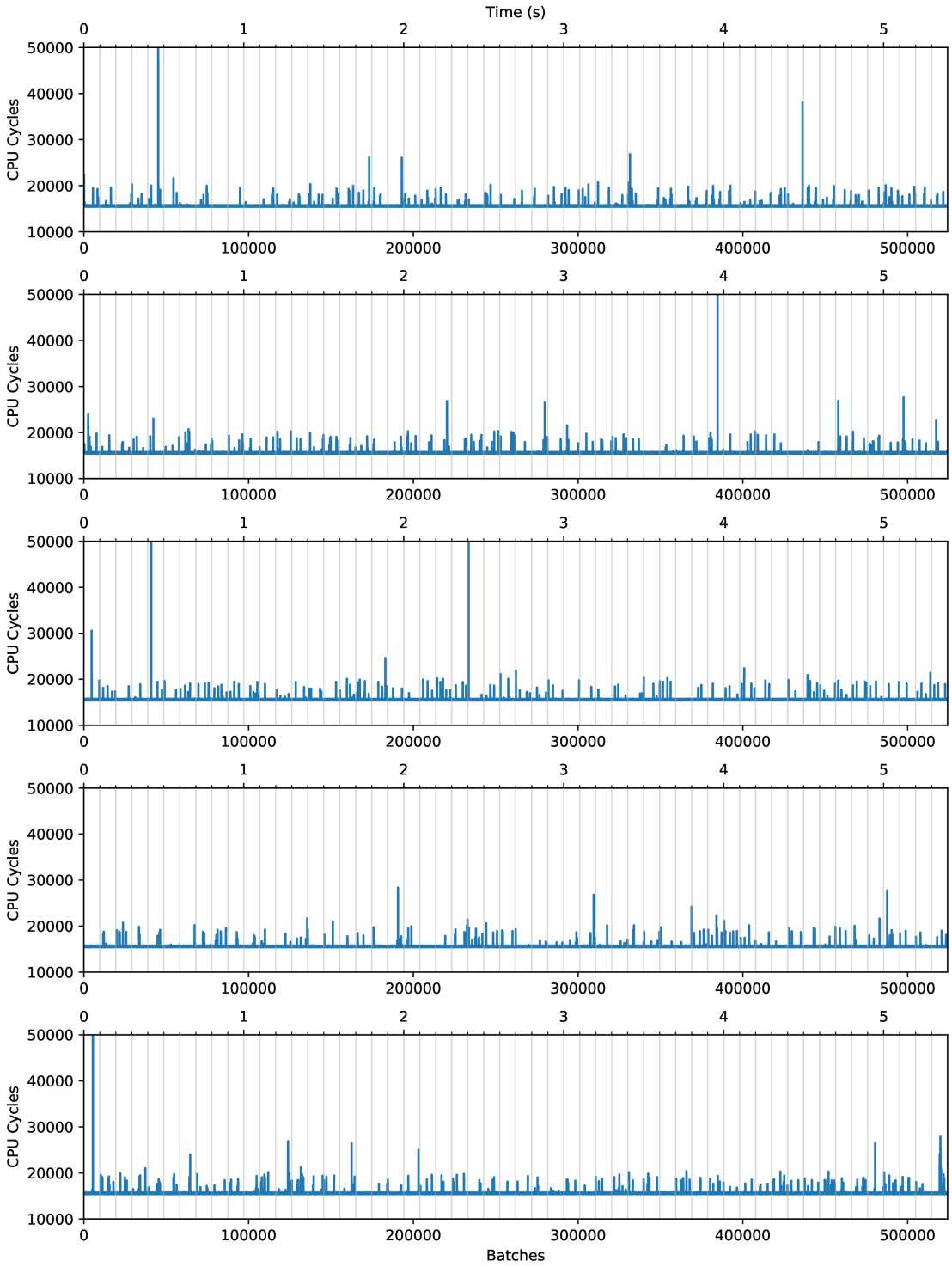


Figure 6.1: CPU cycles per batch of transmitted packets.

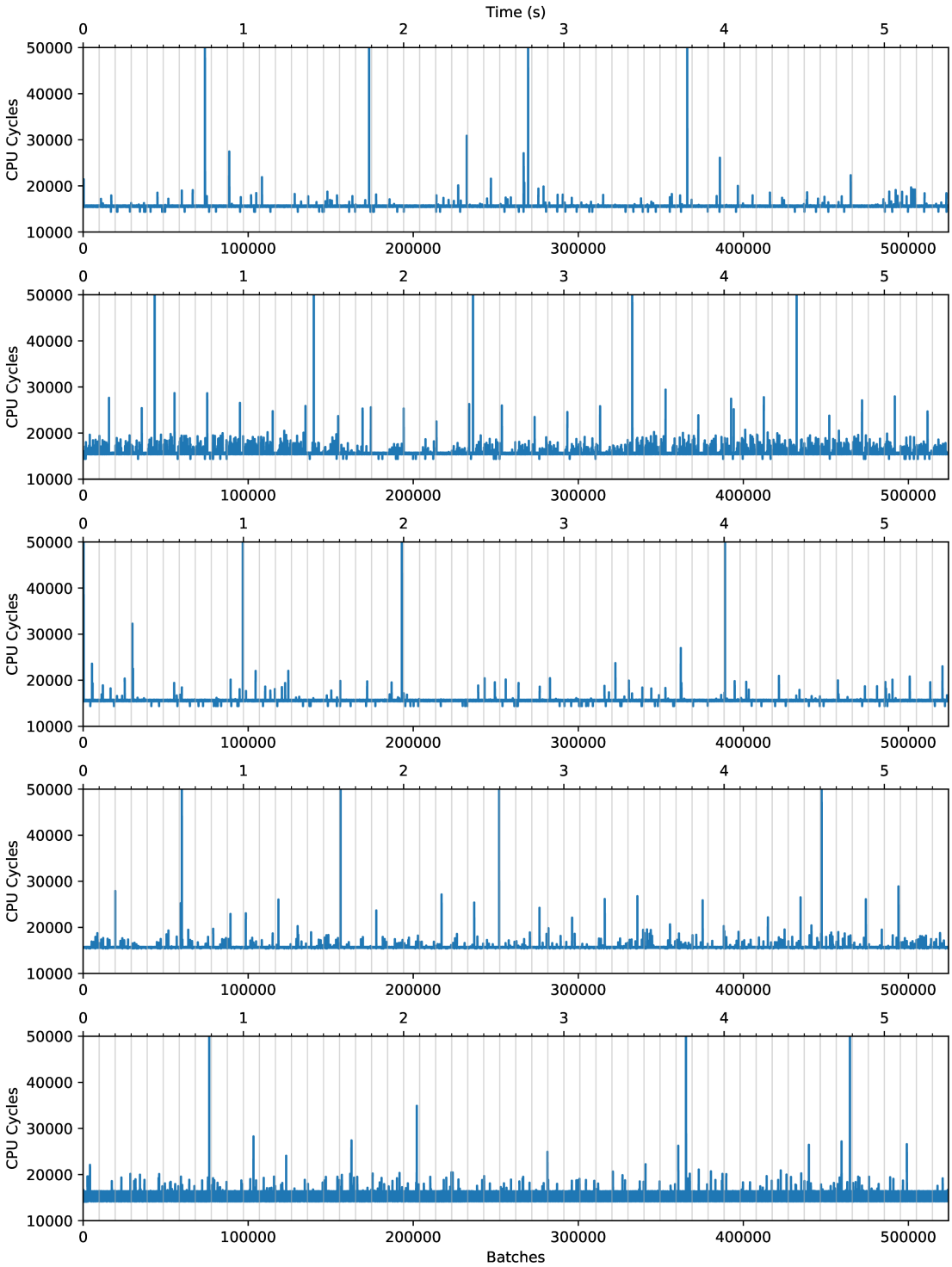


Figure 6.2: CPU cycles per batch of transmitted packets with IOMMU.

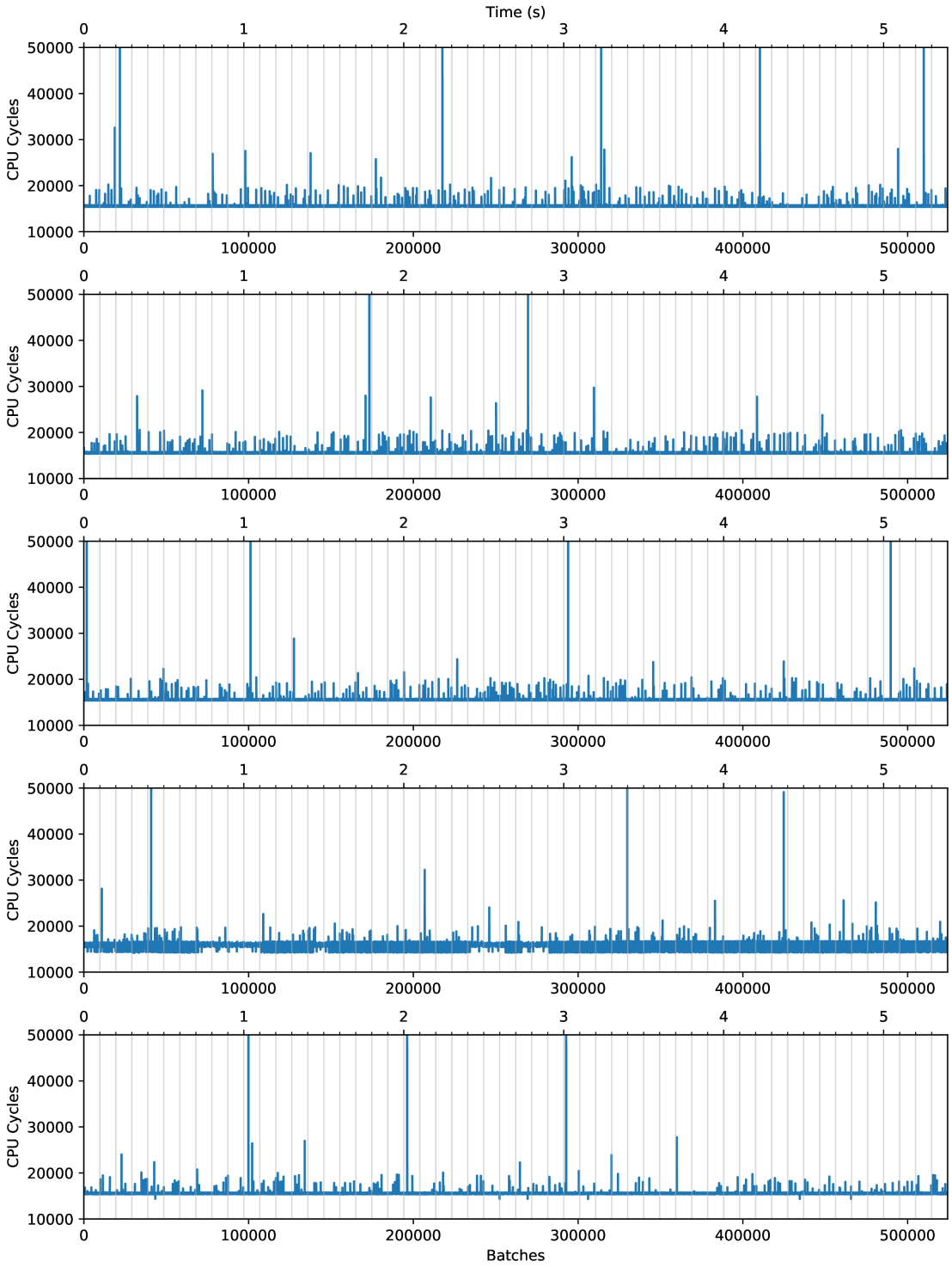


Figure 6.3: CPU cycles per batch of transmitted packets with IOMMU and fixed virtual and physical addresses.

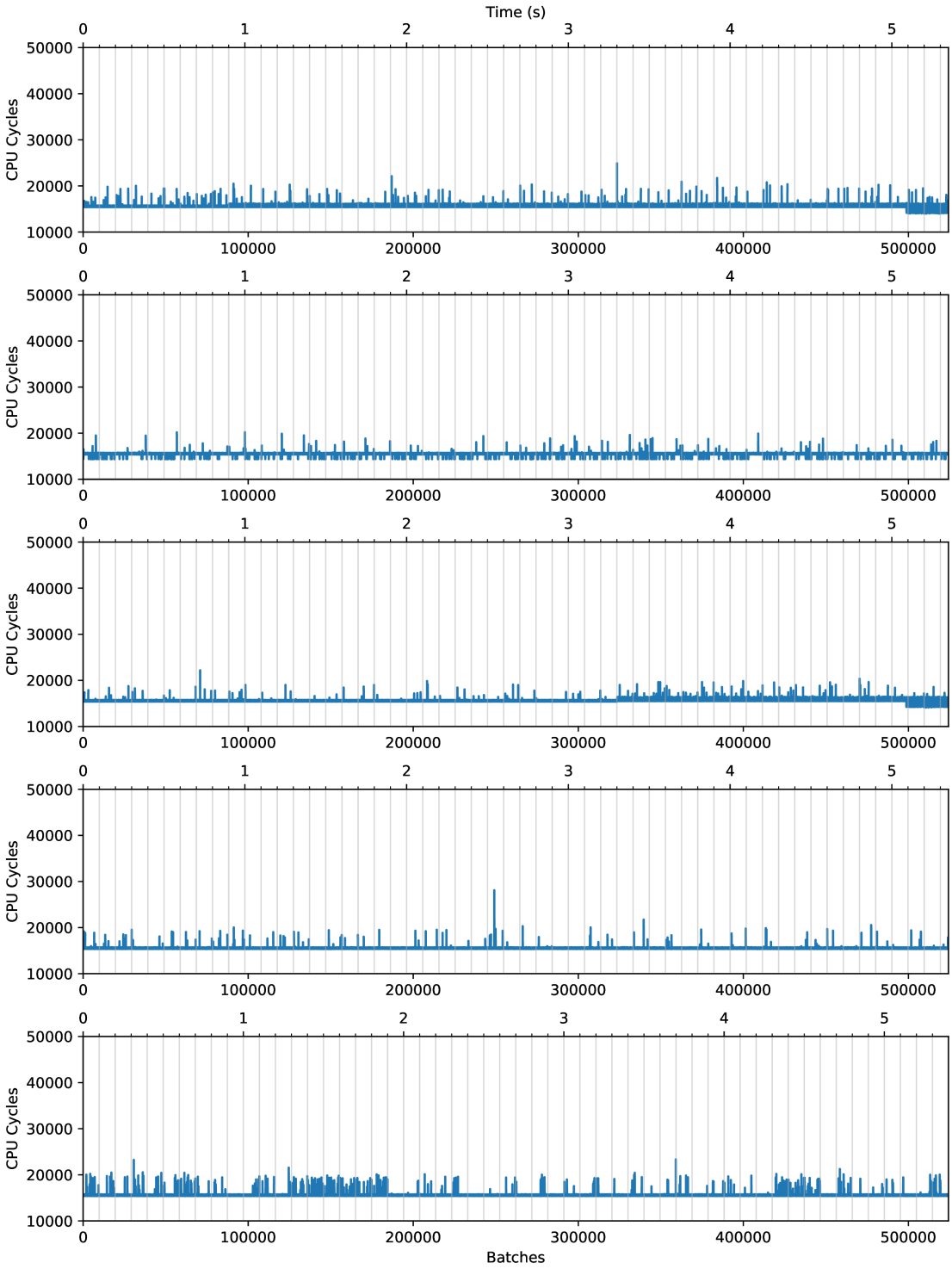


Figure 6.4: CPU cycles per batch of transmitted packets with IOMMU and fixed virtual and physical addresses, other PCIe devices unbound.

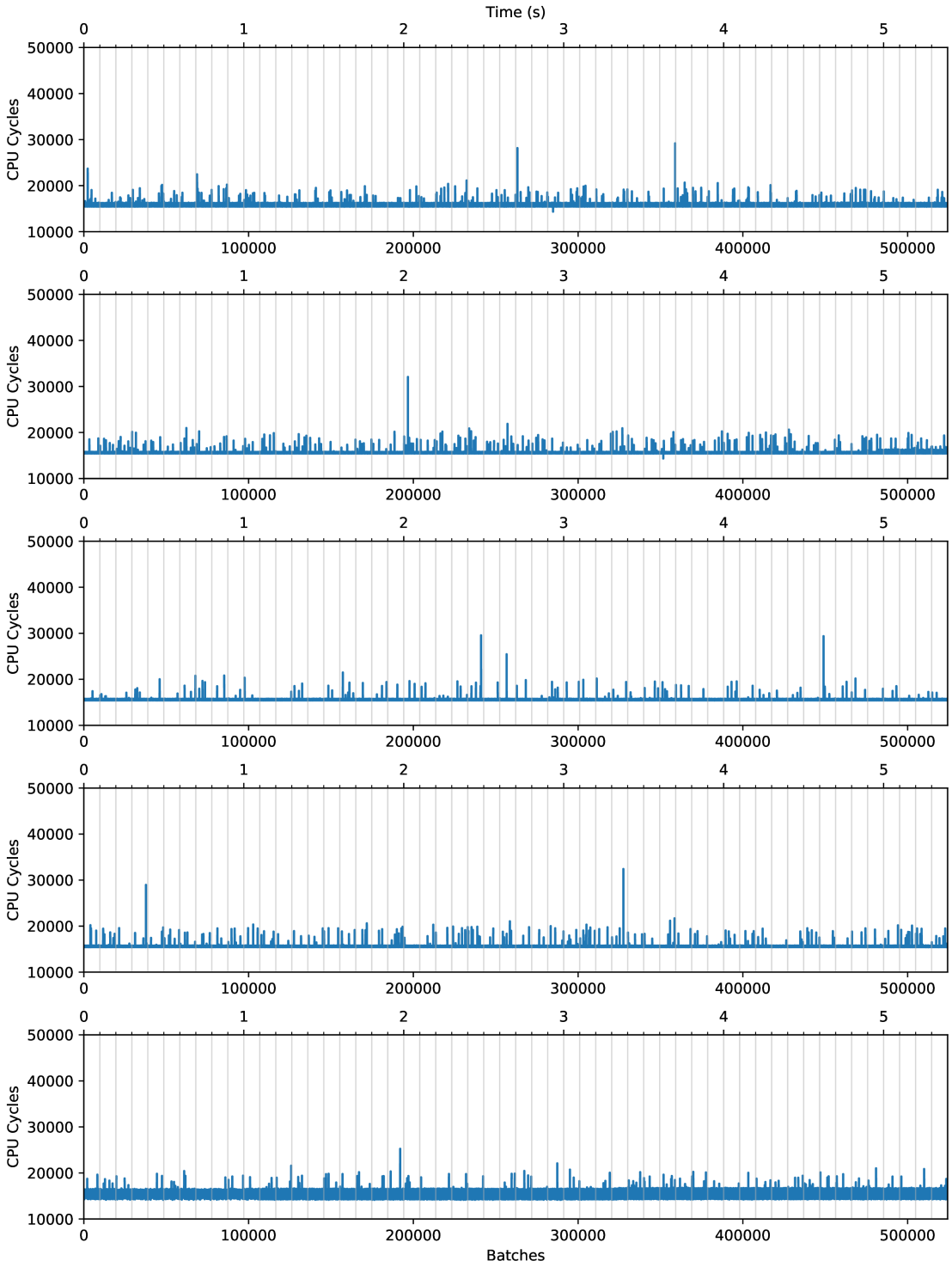


Figure 6.5: CPU cycles per batch of transmitted packets with IOMMU and fixed virtual and physical addresses, other PCIe devices unbound, ping on second NIC port every 0.4 seconds.

We repeat our measurements with the IOMMU in pass-through mode, i.e., our device uses the IOMMU. Figure 6.2 shows the results. Notably, variance of the CPU cycles is even worse with IOMMU. However, we can also notice more patterns in the spikes. In the second run from the top, spikes between 22k and 30k CPU cycles appear every 0.2 s, and spikes above 50,000 CPU cycles every second.

On average, we note 15,682 CPU cycles and a standard deviation of 658 cycles for the runs in Figure 6.2. With a batch size of 63 packets, we get 249 cycles per packet. Thus, we put the cost of the IOMMU at 60 cycles per batch of packets or about 1 cycle per packet.

To minimize variance between runs, we modify our application such that the same virtual and physical memory addresses are used for every run. We allocate a chunk of physical memory on boot using the Linux kernel's `memmap` flag and `mmap` the memory chunk into the process address space via `/dev/mem`.

Figure 6.3 shows the results of running our application with IOMMU and fixed addresses. To our disappointment, CPU cycles still vary greatly although all runs were executed on the same CPU core with the same memory addresses and no dynamic overclocking. Again, we note a periodicity of the peaks of 1.0 and 2.0 s.

As a final attempt, we unbind all PCIe devices not needed for our measurements and repeat our measurements with fixed memory addresses. Figure 6.4 shows that without these PCIe devices, background noise is reduced significantly. However, our measurements still contain too much variance to draw conclusions about IOTLB misses. We verify this assumption by binding the second part of the NIC to the kernel `ixgbe` driver and sending out ping packets every 0.4 s. Figure 6.5 depicts the results of the same measurements with ping running on the second port of the NIC. The plots confirm our assumption. We cannot determine the ping packets with our measurements.

6.3 Summary

IOMMUs provide limited protection against malicious or faulty peripherals due to weaknesses at the PCIe protocol layer. Besides, architectural flaws, missing support and incorrect handling of IOMMUs by operating systems further reduces the safety / security benefits of employing IOMMUs.

Although we suspect the possiblity of side-channel attacks using the IOTLB, our timing measurements in Section 6.2 do not reveal any significant differences between non-IOMMU and IOMMU usage. Our measurements prove to have far too much variance to detect any individual IOTLB misses.

Chapter 7

Conclusion

We investigated the effects of hardware isolation in high-speed network environments with a focus on Intel's and AMD's IOMMUs. We used the `ixy.rs` user space network driver for our analyses, and adapted it to our needs. We implemented support for multiple devices in an IOMMU group and IOMMUs with limited second-level IOVA translation capabilities. We also implemented a whole new driver in `ixy.rs` for virtual functions, `ixgbevf`. And we implemented methods for precise timing measurements on NIC operations to investigate the IOTLB of IOMMUs, including a brute-force memory allocator to allocate physically contiguous memory in user space.

Our findings with `ixy.rs` paint a multi-layered picture of IOMMUs. In regards to performance, our results show that on most configurations, neither throughput nor latency are affected by IOMMUs to a quantifiable extent, be it in virtualized or non-virtualized environments. There are, however, some configurations where IOMMUs have a negative impact, leading to a loss of packet throughput of more than 50%.

In regards to safety and security, our picture of IOMMUs is rather bleak: Contrary to general belief, IOMMUs do not protect against malicious or faulty peripherals due to the weaknesses of the PCIe protocol and archaic threat models of OS vendors which see internal I/O peripherals as trustworthy and enable inherently unsafe features like ATS on these devices.

As a result, many weaknesses in the handling of IOMMUs have been uncovered in the past, as well as some architectural deficiencies. We believe that more vulnerabilities are to be discovered and suspect one in the hardware of IOMMUs: the IOTLB. As other caches, we believe that the IOTLB is susceptible to timing attacks and thus provides an attack surface for side channel attacks. If timing of DMA accesses and

address translation through the IOTLB can be measured precisely, packet reception and transmission times of other devices sharing the IOTLB could be leaked.

Our measurements on DMA operations provide some insight on the performance characteristics of Intel's IOMMU. We hope these results can be used in the future to uncover the suspected vulnerability in the IOTLB.

Generally speaking of safety and security, however, we recognize that a lot has changed in the last years: More and more OSes support IOMMUs or improve their usage of IOMMUs by mitigating vulnerabilities, more vendors of consumer products include IOMMUs into their hardware to protect systems against external peripherals, and even on mobile phones IOMMUs are on the rise: Apple has developed its own IOMMU for the iPhone, Google uses an IOMMU on the Pixel phones, and ARM-based Android phones often have an ARM SMMU included.

In view of these rapid changes, we believe that effects of hardware isolation remain an exciting research topic in all kinds of environments, not necessarily restricted to virtualization and high-speed networking.

7.1 Future Work

Future work could continue our work on the IOTLB, investigate IOMMUs on mobile platforms, e.g., the IOMMU of Apple's iPhones, or focus on Linux's VFIO API and its different ways of handling IOVA address mappings, e.g., coalescing of addresses and mapping with different page sizes.

List of Figures

List of Tables

List of Listings

Chapter A

List of acronyms

ACPI	Advanced Configuration and Power Interface. Open standard for discovery and configuration of computer hardware components by OSes.
BDF	Bus/device/function. Identifier triple of PCIe devices.
BIOS	Basic Input/Output System. Firmware that initializes a computer's hardware on boot before the OS is loaded.
CPU	Central processing unit. Electronic circuitry that executes arithmetic, logic, control and I/O instructions specified by computer programs.
DMA	Direct memory access. Feature of computer systems that allows hardware to access main memory independent of the CPU.
DMAR	DMA remapping. Component of IOMMUs responsible for address translation. Often used synonymously with IOMMU.
I/O	Input/output. Flow of information between an information processing system and a human or another information processing system.
IOMMU	Input-output memory management unit. A MMU for I/O devices.
IOTLB	I/O translation lookaside buffer. TLB of an IOMMU.
IOVA	I/O virtual address. Address used by I/O devices for DMA access. IOMMUs translate IOVAs to physical adresses.
MAC	Medium access control. Sublayer of the data link layer responsible for flow control and multiplexing of the transmission medium.
MMU	Memory management unit. Computer hardware component that primarily translates virtual to physical memory addresses. Often part of the CPU.

NFV	Network function virtualization. Creation of building blocks by virtualizing network functionality traditionally implemented in hardware (e.g., NAT, firewalls), and chaining of virtualized blocks to create communication services.
NIC	Network interface controller. Computer hardware component that receives and transmits data via a computer network.
OS	Operating system. Low-level software that manages a computer's hardware and software and provides services for computer programs.
PCIe	Peripheral Component Interconnect Express. High-speed serial bus standard designed to replace PCI.
PF	Physical function. Fully featured PCIe function that supports SR-IOV. Creates and manages VFs and shared resources such as the link.
SDN	Software-defined networking. Approach to dynamic network management by splitting switches into remotely configurable control planes (routing information) and data planes (packet forwarding).
SMMU	System memory management unit. ARM's term for IOMMU.
SR-IOV	Single root input/output virtualization. Feature of computer systems that allows a single PCIe device to be shared with multiple virtual environments.
TLB	Translation lookaside buffer. Memory cache of a MMU that stores recent address translations to improve performance.
UEFI	Unified Extensible Firmware Interface. Standardized successor of the BIOSes found in IBM PC-compatible computers.
VF	Virtual function. Lightweight PCIe function of SR-IOV-capable devices that is associated with a PF and shares resources with its PF and other VFs. Often passed-through to virtual environments.
VFIO	Virtual Function I/O. Linux driver framework that utilizes IOMMUs and allows direct device access from user space in a safe manner.

Bibliography

[1] Keith Adams and Ole Agesen. "A Comparison of Software and Hardware Techniques for x86 Virtualization". In: *ACM Sigplan Notices* 41.11 (2006), pp. 2–13.

[2] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. "IOMMU: Strategies for mitigating the IOTLB bottleneck". In: *International Symposium on Computer Architecture*. 2010, pp. 256–274.

[3] Nadav Amit et al. "vIOMMU: Efficient IOMMU Emulation". In: *USENIX Annual Technical Conference (ATC)*. 2011, pp. 73–86.

[4] Michael Becher, Maximillian Dornseif, and Christian N Klein. "FireWire: all your memory are belong to us". In: *Proceedings of CanSecWest* (2005), p. 67.

[5] Muli Ben-Yehuda et al. "The Price of Safety: Evaluating IOMMU Performance". In: *The Ottawa Linux Symposium*. 2007, pp. 9–20.

[6] Muli Ben-Yehuda et al. "Utilizing IOMMUs for virtualization in Linux and Xen". In: *OLS'06: The 2006 Ottawa Linux Symposium*. 2006, pp. 71–86.

[7] Gal Beniamini. *Over The Air - Vol. 2, Pt. 3: Exploiting The Wi-Fi Stack on Apple Devices*. `https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-3-exploiting-wi-fi.html`. Accessed: 2021-03-11.

[8] Roy Chua. *State of SDN and NFV — Hype or Reality?* `https://www.sdxcentral.com/articles/analysis/state-of-sdn-and-nfv-hype-or-reality/2018/04/`. Accessed: 2021-04-05.

[9] Levente Csikor et al. "Transition to SDN is HARMLESS: Hybrid Architecture for Migrating Legacy Ethernet Switches to SDN". In: *IEEE/ACM Transactions on Networking* 28.1 (2020), pp. 275–288.

[10] Marion Daubignard and Yves-Alexis Perez. "ProTIP: You Should Know What to Expect From Your Peripherals". In: *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*. 2017.

[11] Simon Ellmann. *GitHub Repository: iommu-leaks*. `https://github.com/ackxolotl/iommu-leaks`. Accessed: 2021-04-08.

[12] Simon Ellmann. *GitHub Repository: ixy.rs*. https://github.com/ixy-languages/ixy.rs/. Accessed: 2021-04-08.

[13] Simon Ellmann. "Writing Network Drivers in Rust". BA thesis. Technical University Munich, 2018.

[14] Paul Emmerich et al. "Moongen: A Scriptable High-Speed Packet Generator". In: *Proceedings of the 2015 Internet Measurement Conference*. 2015, pp. 275–287.

[15] Paul Emmerich et al. "The Case for Writing Network Drivers in High-Level Programming Languages". In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2019, pp. 1–13.

[16] Paul Emmerich et al. "User Space Network Drivers". In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2019, pp. 1–12.

[17] heftig. *Arch Linux Commit: 'Disable INTEL_IOMMU_DEFAULT_ON'*. https://git.archlinux.org/svntogit/packages.git/commit/?h=packages/linux&id=2e42789fe089b87dfe6763e7d48781b6a0c33b95. Accessed: 2021-03-17.

[18] Stefan Huber. "Using the IOMMU for Safe and Secure User Space Network Drivers". MA thesis. Technical University Munich, 2019.

[19] Ben Hutchings. *Debian Commit: 'Enable INTEL_IOMMU_DEFAULT_ON'*. https://salsa.debian.org/kernel-team/linux/-/commit/d6f56e293f5e8d5afacda5eb4d63c9bc8c860446. Accessed: 2021-03-17.

[20] *Intel 82599 10 GbE Controller Datasheet Rev. 3.4*. Intel. Nov. 2019.

[21] *Intel 82599 SR-IOV Driver Companion Guide Rev. 1.00*. Intel. May 2010.

[22] *Intel Virtualization Technology for Directed I/O Rev. 3.1*. Intel. June 2019.

[23] Sushant Jain et al. "B4: Experience with a Globally-Deployed Software Defined WAN". In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 3–14.

[24] Michael Kistler, Michael Perrone, and Fabrizio Petrini. "Cell Multiprocessor Communication Network: Built for Speed". In: *IEEE micro* 26.3 (2006), pp. 10–23.

[25] Ryota Kunisawa. "Gigabit Network with Cooperative Functions for General Purpose Massively Parallel OS". MA thesis. University of Tokyo, 1997.

[26] Daniel Lemire. *AMD Zen 2 and branch mispredictions*. https://lemire.me/blog/2019/12/06/amd-zen-2-and-branch-mispredictions/. Accessed: 2021-02-14.

[27] Daniel Lemire. *Instructions per cycle: AMD Zen 2 versus Intel*. https://lemire.me/blog/2019/12/05/instructions-per-cycle-amd-versus-intel/. Accessed: 2021-02-14.

[28] Moshe Malka, Nadav Amit, and Dan Tsafrir. “Efficient Intra-Operating System Protection Against Harmful DMAs”. In: *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. 2015, pp. 29–44.

[29] Moshe Malka et al. “rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers”. In: *ACM SIGPLAN Notices* 50.4 (2015), pp. 355–368.

[30] Theo Markettos et al. “Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals”. In: (2019).

[31] Kyle McMartin. *Linux Kernel Commit: 'x86: disable intel_iommu support by default'*. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0cd5c3c80a0ebd68c08312fa7d8c13149cc61c4c`. Accessed: 2021-03-17.

[32] David S. Miller and Miguel de Icaza. *The Linux/SPARC port*. 1996.

[33] Benoît Morgan et al. “IOMMU protection against I/O attacks: a vulnerability and a proof of concept”. In: *Journal of the Brazilian Computer Society* 24.1 (2018), pp. 1–11.

[34] Rolf Neugebauer et al. “Understanding PCIe performance for end host networking”. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 327–341.

[35] Gabriele Paoloni. “How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures”. In: *Intel Corporation* 123 (2010).

[36] *PCI Express Base Specification Rev. 4.0 Vers. 1.0*. PCI-SIG. Sept. 2017.

[37] Omer Peleg et al. “Utilizing the IOMMU Scalably”. In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015, pp. 549–562.

[38] Colin Lewis Rothwell. “Exploitation from malicious PCI Express peripherals”. PhD thesis. University of Cambridge, 2018.

[39] Abhishek Verma et al. “Large-scale cluster management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–17.

[40] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. “Another Way to Circumvent Intel Trusted Execution Technology”. In: *Invisible Things Lab* (2009), pp. 1–8.