Fakultät für Elektrotechnik und Informationstechnik
Technische Universität München

Mitigating and Resolving Performance Isolation Issues of PCIe Passthrough and SR-IOV in Multi-Core Virtualization

Andre Oliver Richter

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:
Prof. Dr.-Ing. Wolfgang Kellerer

Prüfende der Dissertation:
1. Prof. Dr. sc. techn. Andreas Herkersdorf
2. Prof. Dr.-Ing. Dr. h. c. Jürgen Becker,
   Karlsruher Institut für Technologie (KIT)

Die Dissertation wurde am 24.04.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 27.10.2017 angenommen.

Danksagung

Zuallererst möchte ich meinem Doktorvater Prof. Dr. sc. techn. Andreas Herkersdorf für die Betreuung dieser Arbeit danken. Er hat es mir ermöglicht an einem spannenden und für die weitere Zukunft sehr relevanten Thema zu forschen und damit den Grundstein für diese Arbeit gelegt. Seine Ideen in technischen Meetings halfen mir, sehr schnell ein passendes Gebiet zu identifizieren, auf dem ich forschen konnte. Hervorheben möchte ich auch Prof. Herkersdorfs menschliche Führung, die seinen Lehrstuhl zu einem angenehmen und vertrauensvollen Arbeitsumfeld für mich machte.

Besonderer Dank gilt auch Prof. Dr.-Ing. Dr. h. c. Jürgen Becker als Zweitprüfer dieser Arbeit und Prof. Dr.-Ing. Wolfgang Kellerer für den Vorsitz der Prüfungskommission.

Weiterer Dank gilt:

Dr.-Ing. Thomas Wild für seinen Input in technischen Meetings und seine Hilfe bei organisatorischen Dingen aller Art, vor allem im ARAMiS Projekt. Außerdem für die Möglichkeit der Tutorentätigkeit im Praktikum VHDL und die dortige Betreuung, durch die ich den Lehrstuhl schon als Student kennen lernen konnte.

Christian Herber, mit dem ich über die Jahre stets effektiv zusammen an diversen Publikationen und im ARAMiS Projekt gearbeitet habe und bei unseren Dienstreisen immer eine gute Zeit hatte.

Holm Rauchfuss, der beigetragen hat, dass der Lehrstuhl Partner im ARAMiS Projekt wurde. Dadurch konnte die Stelle geschaffen werden, auf der ich eingestellt wurde. Des Weiteren für die Betreuung während meiner Werkstudententätigkeit, welche meine erste Arbeitsstelle am Lehrstuhl war.

Allen anderen Kollegen für die gute Zeit, die lockere Atmosphäre, den guten Umgang, die vielen Dart Matches und natürlich die hitzigen Debatten (nicht zwangsweise immer nur über Forschung).

Meiner Frau Susanne, die mich immer aufs Neue motiviert hat, und vor allem gegen Ende eine große Unterstützung in jeglicher Hinsicht war. Zu guter Letzt meinen Eltern, für ihre Geduld und ihr Vertrauen während der Schulzeit und die bedingungslose Unterstützung während dem Studium. Ihr habt diesen Weg geebnet.

Zusammenfassung

Leistungsisolierung ist eine zentrale Herausforderung in virtualisierten Multicore Systemen. Sie beschreibt die Fähigkeit eines Wirtsystems zur Verhinderung oder Limitierung von Leistungsinterferenzen zwischen mehreren Virtuellen Maschinen (VMs), welche auf einem Wirtsystem ausgeführt werden und sich dessen Hardware-Ressourcen teilen. Ist nur unzureichende Isolierung geboten, wird es möglich, dass bösartige oder defekte VMs Leistung von nebenläufigen VMs oder dem Wirtsystem selbst stehlen oder vermindern. In der Cloud-Computing Domäne, in der virtualisierte Multicore Server dem aktuellen Stand der Technik entsprechen, kann dies zu Vertragsverletzungen zwischen Kunden und Anbieter führen. In der Domäne für eingebettete Systeme, in welcher virtualisierte Multicore Systeme in naher Zukunft auftauchen werden, können Probleme mit der Leistungsisolierung sogar zur Verletzung von Realzeitanforderungen führen, was letztendlich die Systemsicherheit gefährden könnte. Da virtualisierte Systeme viele gemeinsam-benutzte Ressourcen verwenden, z.B. CPU-Kerne, Caches oder I/O Geräte, muss Leistungsisolierung immer individuell bewertet werden.

Diese Arbeit präsentiert Forschungsbeiträge zur Leistungsisolierung der neusten Generation von I/O Virtualisierungstechnologie, namentlich PCIe passthrough und Single Root I/O Virtualization (SR-IOV). Diese Virtualisierungstechnologie ermöglicht niedrige Latenzen und hohe Leistung, und ist bereits in kommerziellen Geräten erhältlich. Die Arbeit zeigt auf, dass die Leistungsisolierung von PCIe passthrough und SR-IOV durch Denial-of-Service (DoS) Angriffe bösartiger oder defekter VMs gebrochen werden kann, indem passthrough und SR-IOV Hardware mit sinnlosen PCIe Paketen überflutet wird. Zum Beispiel führt ein DoS-Angriff auf einen SR-IOV-fähigen Zweiport Gigabit Ethernet Adapter zu einer Verminderung des Durchsatzes von 941 Mbit/s zu 615 Mbit/s (-35%) Derselbe Angriff führt zu einer Verminderung der SSD-Leistung des Wirtsystems um 77%. Diese Arbeit untersucht diesen Angriffsvektor gründlich, steuert eine Klassifikation verschiedener DoS-Angriffstypen bei und präsentiert ein Modell, welches die architekturellen Probleme in aktueller Hardware beschreibt, die die DoS-Angriffe ermöglichen.

Des Weiteren werden zwei domänenspezifische Lösungen präsentiert, welche die entdeckten Isolierungsprobleme abschwächen oder lösen. Die Lösung für die Cloud Domäne verwendet schlanke Hardware-Monitoring Erweiterungen in passthrough Geräten, um DoS-Angriffe zu entdecken und mit Software-Scheduling abzuschwächen. Zum Beispiel ermöglicht sie die Leistung einer Apache Webserver VM von 51.3% während eines DoS-Angriffs zurück auf 97% zu heben. Als Lösung für Embedded Systeme werden zwei integrierte Hardware-Architekturen vorgeschlagen. Diese funktionieren optionale und bis dato vernachlässigte QoS Erweiterungen der PCIe Spezifikation so um, dass der DoS-Angriffsvektor geschlossen wird und Angriffe komplett verhindert werden. Die erste Ar-

chitektur ist für Scheduling-Freiheit optimiert, die zweite für minimale Hardwarekosten. Die Vor- und Nachteile der beiden Architekturen werden anhand von Evaluierungsergebnissen diskutiert.

Nach meinem Wissen sind die Beiträge, welche in dieser Arbeit präsentiert werden, die ersten die eine umfassende Untersuchung von DoS-Angriffen auf PCIe passthrough und SR-IOV anstellen, und Beiträge zur Lösung der entdeckten Leistungsisolierungsprobleme leisten.

Abstract

Performance isolation is a key challenge in virtualized multi-core systems. It describes a host system's capability to prevent or limit performance interference between multiple Virtual Machines (VMs) that execute on it and share its hardware resources. If a host system provides insufficient performance isolation, it is possible for malicious or malfunctioning VMs to degrade or steal performance from concurrent VMs or the host itself. In the cloud computing domain, where virtualized multi-core servers are state-of-the-art, performance interference could result in broken service level agreements between providers and customers. In the embedded computing domain, where virtualized multi-core systems will surface in the near future, performance isolation issues might even result in violation of real-time requirements and therefore compromise system safety. As virtualized systems employ many shared resources, e.g. CPU cores, caches or I/O devices, performance isolation must be assessed individually.

This thesis contributes research on performance isolation of PCIe passthrough and Single Root I/O Virtualization (SR-IOV), which is the latest generation of I/O virtualization technology. It enables low latency, high performance virtual I/O and is already implemented in commercial off-the-shelf devices. The thesis shows that performance isolation of PCIe passthrough and SR-IOV can be broken with Denial-of-Service (DoS) attacks from malicious or malfunctioning VMs that flood passthrough and SR-IOV hardware with spurious PCIe packets. For example, DoS attacks on an SR-IOV capable dual-port gigabit Ethernet adapter cause a throughput degradation from 941 Mbit/s down to 615 Mbit/s (-35%). The same attack causes a 77% performance degradation of the host system's SSD storage. The thesis thoroughly investigates this attack vector, contributes a classification of different DoS attack types and presents a model that explains the architectural shortcomings in current hardware that enable the attacks.

Additionally, two domain specific solutions for cloud and embedded systems are presented, which mitigate and resolve the discovered performance isolation issues. The cloud solution utilizes lightweight hardware monitoring extensions within passthrough I/O devices to detect attacker VMs and mitigate their DoS attacks with software scheduling. For instance, performance of an Apache webserver VM is restored from 51.3% during DoS attacks back to 97%. The solution for embedded systems proposes two integrated hardware architectures that repurpose optional and neglected QoS extensions of the PCIe specification in order to close the attack vector and completely prevent DoS attacks. The first architecture is optimized for scheduling freedom, the second for minimal hardware costs. Results of an evaluation are used to discuss pros and cons of both architectures.

To the best of my knowledge, work presented in this thesis is the first that comprehensively investigates DoS attacks on PCIe passthrough and SR-IOV, and contributes towards solving the resulting performance isolation issues.

Contents

Danksagung iii

Zusammenfassung v

Abstract vii

Contents ix

List of Figures xiii

List of Tables xv

Acronyms xvii

1 Introduction 1
1.1 Problem Statement . . . 2
1.2 Contributions . . . 3
1.3 Outline . . . 5

2 State of the Art 7
2.1 Platform Virtualization . . . 7
2.1.1 Spatial Isolation . . . 8
2.1.2 Performance Isolation . . . 9
2.1.3 Cloud Computing . . . 9
2.1.4 Embedded Virtualization . . . 10
2.2 Software-based I/O Virtualization . . . 11
2.2.1 Emulation . . . 11
2.2.2 Paravirtualization . . . 13
2.2.3 Performance, Pros and Cons . . . 14
2.2.4 Performance Isolation Approaches . . . 15
2.3 Hardware-based I/O Virtualization: Passthrough I/O with PCIe and SR-IOV . . . 16
2.3.1 PCI Express (PCIe) . . . 17
2.3.1.1 Topology . . . 17
2.3.1.2 Link Parameters and Bit Rates . . . 18
2.3.1.3 PCIe Functions, Memory Mapped I/O and Drivers . . . 18
2.3.1.4 Transaction Characteristics and Flow Control . . . 20

2.3.2 PCIe Single Root I/O Virtulization (SR-IOV) 20
2.3.2.1 Physical and Virtual Functions 20
2.3.2.2 Adoption 22
2.3.3 Performance, Pros and Cons 23
2.3.4 Performance Isolation Approaches 23

3 Investigation of PCIe Passthrough and SR-IOV Performance Isolation Issues 27
3.1 Threat Model 27
3.2 Implementation of DoS Attacks 28
3.3 Evaluation Platform 29
3.4 Evaluation Methodology 31
3.4.1 Latencies of 32 bit PCIe Read Transactions 31
3.4.2 Network Performance – TCP and UDP Throughput 33
3.4.3 Storage Performance – SSD Throughput 33
3.5 Experiments and Results 33
3.5.1 Experiment 1: Attacking SR-IOV Virtual Functions 34
3.5.1.1 Baseline Performance 34
3.5.1.2 Experimental Setup and Results 34
3.5.1.3 Second-Order Effects 36
3.5.1.4 Summary 37
3.5.2 Experiment 2: Influence of the DoS Victim's Processing Speed . . 37
3.5.2.1 Fine-Grained Runtime-Configurable Processing Speeds via FPGA 38
3.5.2.2 Experimental Setup and Results 38
3.5.2.3 Processing Speed Variations in COTS Devices 41
3.5.2.4 Summary 42
3.5.3 Experiment 3: Influence of Switching Intermediaries 43
3.5.3.1 Experimental Setup and Results 43
3.5.3.2 Summary 43
3.5.4 Experiment 4: Degradation of Chipset Disk I/O 45
3.5.4.1 Experimental Setup and Results 45
3.5.4.2 Summary 46
3.6 System Model for PCIe DoS Attacks on Ethernet NICs 47
3.7 DoS Attack Classification 49
3.7.1 Inter-Device Attack 50
3.7.2 Intra-Device Attacks 51
3.7.2.1 Inter-Function Attack 51
3.7.2.2 Intra-PF Attack 51
3.7.2.3 Inter-PF Attack 52
3.7.3 Combinations 53
3.8 Summary 53

4 Mitigating Performance Isolation Issues Through Monitoring and Scheduling 55
4.1 Goals and Requirements . . . 55
4.1.1 Goals . . . 55
4.1.2 Requirements . . . 56
4.2 Design and Exploration . . . 57
4.2.1 Monitoring and DoS Attack Detection . . . 57
4.2.2 Exploration of Monitoring Alternatives . . . 58
4.2.2.1 Monitoring in CPUs . . . 58
4.2.2.2 Monitoring in I/O Devices . . . 60
4.2.2.3 Monitoring in Intermediaries . . . 61
4.2.3 Mitigation . . . 61
4.2.3.1 Isolation via Freeze or Migration . . . 61
4.2.3.2 Attack Throttling via Scheduling . . . 62
4.3 Implementation . . . 63
4.3.1 Overview . . . 63
4.3.2 Monitoring and DoS Detection on FPGA . . . 64
4.3.3 VC709 PF Driver and DoS Protect Process . . . 65
4.3.4 System Software and Emulation of Monitoring in the 82576 . . . 66
4.4 Evaluation . . . 67
4.4.1 Benchmarks . . . 68
4.4.2 DoS Attack Parameters . . . 68
4.4.3 DoS Detection Threshold . . . 69
4.4.4 Memcached and Apache Results . . . 70
4.4.5 Netperf Results . . . 72
4.4.6 Trace of DoS Detection and Attacker Freeze . . . 74
4.4.6.1 Baseline Trace . . . 74
4.4.6.2 Mitigation Trace . . . 76
4.4.6.3 Optimization Tradeoffs . . . 76
4.4.7 Mirroring Overhead . . . 76
4.4.7.1 Benchmark Overhead . . . 77
4.4.7.2 DoS Attack Overhead . . . 77
4.5 Summary . . . 78

5 Resolving Performance Isolation Issues Through PCIe QoS Extensions 81
5.1 Goals and Requirements . . . 82
5.1.1 Goals . . . 82
5.1.2 Requirements . . . 82
5.2 Quality-of-Service Extensions in the PCIe Specification . . . 83
5.2.1 Contribution of this Thesis . . . 83
5.2.2 Overview of the PCIe Specification's QoS Extensions . . . 84
5.2.2.1 TC/VC Mapping . . . 85
5.2.2.2 Arbitration . . . 86
5.3 SystemC TLM 2.0 Model of QoS-enabled PCIe Components . . . 87
5.3.1 Model Verification Approach . . . 90

5.3.2 Verification Results 90
5.4 Multi-VC Architecture 93
5.4.1 Architecture 93
5.4.2 Evaluation 95
5.4.3 Multi-VC Architecture Summary 97
5.5 Time-based Weighted Round-Robin Architecture 98
5.5.1 Architecture 98
5.5.2 Evaluation 100
5.5.3 TBWRR Architecture Summary 102
5.6 Summary 102

6 Conclusion and Outlook **105**
6.1 Conclusion 105
6.2 Outlook 108

Bibliography **111**

A Approximating Packet Processing Times of COTS PCIe Devices **123**

List of Figures

1.1 Overview of contributions 4

2.1 Comparison of classic hypervisor types 8
2.2 I/O device emulation in a type 1 hypervisor 11
2.3 I/O device emulation with type 1 hypervisor and trusted VM 12
2.4 I/O device emulation in hosted hypervisors 13
2.5 Paravirtualized I/O with trusted VM 14
2.6 Example of passthrough I/O 16
2.7 Example PCIe topology 17
2.8 PCIe multi-function Endpoint 19
2.9 Virtualized system with SR-IOV device 21
2.10 Dependency between interconnect and I/O functionality 24

3.1 Block diagram of passthrough/SR-IOV evaluation platform 30
3.2 TSC virtualization side effects 32
3.3 Impact of DoS attacks on VF0.0 35
3.4 Influence of PCIe speed on performance degradation 36
3.5 Influence of packet processing speed on latency during DoS 39
3.6 Perf. degradation depending on DoS victim's processing speed 40
3.7 PCIe device multiple processing speeds 41
3.8 Influence of switching intermediaries on performance degradation 44
3.9 Impact of DoS attacks on chipset disk I/O 46
3.10 System model for legal software-NIC communication 47
3.11 System model for software-NIC communication with a malicious VM . . . 49
3.12 Inter-device attack 50
3.13 Inter-function attack 51
3.14 Intra-PF attack 52
3.15 Inter-PF attack 52
3.16 DoS attack combinations 53

4.1 High-level design of HW/SW monitoring/mitigation design 57
4.2 Overview of x86 performance monitoring facilities 59
4.3 Design for DoS detection in I/O devices 60
4.4 Throttling malicious VMs via scheduling 62
4.5 Prototype implementation for mitigating DoS attacks 63
4.6 Implementation of hardware monitoring and DoS detection on the Virtex-7 FPGA 64
4.7 PCIe transaction rates – Legal vs. DoS 70

4.8 Memcached results . . . 71
4.9 Apache results . . . 71
4.10 Netperf results for TCP and UDP . . . 73
4.11 A trace of DoS attack mitigation by freezing the attacker . . . 75

5.1 Relationship between PCIe, its optional QoS extensions and SR-IOV . . . 84
5.2 PCIe QoS extensions . . . 85
5.3 TBWRR arbitration table example . . . 86
5.4 Block diagram of the real-world lab-setup . . . 88
5.5 The PCIe hierarchy of the SystemC model . . . 89
5.6 Comparison: Lab-setup vs. SystemC model . . . 91
5.7 Multi-VC architecture . . . 94
5.8 UDP results for multiple, concurrent DoS attacks . . . 96
5.9 Comparison of processing engine designs for SR-IOV devices . . . 97
5.10 TBWRR architecture . . . 99
5.11 TBWRR table tailored to the 82576 NIC . . . 99
5.12 TBWRR vs. best effort arbitration . . . 100
5.13 TBWRR table with idle cores . . . 101
5.14 TBWRR table with two stage arbitraion . . . 102

6.1 Characteristics of the cloud computing solution . . . 107
6.2 Characteristics of the solutions for embedded virtualization . . . 107

A.1 Three phases of a DoS attack. . . . 123

List of Tables

2.1 Performance parameters by PCIe version. 18

3.1 Transfer times for a 1500 byte Ethernet payload. 37

4.1 Latencies for reading from a 82576 NIC VF with and without concurrent DoS attack on VF0.0. 69
4.2 Times for executing a for-loop (DoS attack) with 10^8 consecutive PCIe write transactions, depending on device. 77

5.1 Two examples for TC/VC mapping for a port with two VCs. 85
5.2 Downstream PCIe packet sources classified by trust. 93

Acronyms

API	Application Programming Interface.
ARI	Alternative Routing-ID Interpretation.
ARINC	Aeronautical Radio, Incorporated.
ASIC	Application-Specific Integrated Circuit.
BAR	Base Address Register.
BIOS	Basic Input/Output System.
CAN	Controller Area Network.
COTS	Commercial off-the-shelf.
CPU	Central Processing Unit.
DMA	Direct Memory Access.
DMI	Direct Media Interface.
ECU	Electronic Control Unit.
EPT	Extended Page Tables.
FPGA	Field-Programmable Gate Array.
GPU	Graphics Processing Unit.
I/O	Input/Output.
IDD	Integrated Driver Domain.
IOMMU	Input/Output Memory Management Unit.
IP	Internet Protocol.
IRQ	Interrupt Request.
KVM	Kernel Virtual Machine.
LLC	Last Level Cache.
MDD	Malicious Driver Detection.
MMIO	Memory Mapped I/O.
MMU	Memory Management Unit.
MTU	Maximum Transmission Unit.

NIC	Network Interface Card.
NUMA	Non-Uniform Memory Access.
OS	Operating System.
PCH	Platform Controller Hub - The chipset of a computer.
PCI	Peripheral Component Interconnect.
PCI-SIG	PCI Special Interest Group.
PCIe	PCI Express.
PF	Physical Function.
PID	Process ID.
PMC	Performance Monitoring Counter.
PMU	Performance Monitoring Unit.
QEMU	Short for Quick Emulator - An open-source hypervisor.
RAM	Random Access Memory.
RR	Round-Robin.
SR-IOV	Single Root I/O Virtualization.
SSD	Solid-State Drive.
TBWRR	Time-based Weighted Round-Robin.
TC	Traffic Class.
TCP	Transmission Control Protocol.
TSC	Time-Stamp Counter.
UDP	User Datagram Protocol.
UIO	Userspace I/O.
VC	Virtual Channel.
VF	Virtual Function.
VHDL	Very High Speed Integrated Circuit Hardware Description Language.
VM	Virtual Machine.
WCET	Worst-Case Execution Time.
WRR	Weighted Round-Robin.

1 Introduction

Virtualization of multi-core server systems is a key enablement technology for the modern cloud computing landscape [1]. Using the virtualization capabilities of modern CPUs [2, 3], it is possible to encapsulate whole Operating Systems (OSs), together with user-applications that run on top of these OSs, into so-called Virtual Machines (VMs). Multiple VMs can execute concurrently on the same physical server and share its hardware, therefore making virtualization an effective method for increasing the utilization of today's multi-core servers. As VMs can be generated, destroyed and migrated on demand, they also introduce flexibility to datacenter operators. These features are also important from an economic perspective. On the one hand, they enable cloud computing providers to boost the efficiency of their expensive hardware by consolidating VMs of multiple customers on a single physical machine. On the other hand, they allow providers to introduce fine-grained and flexible pricing models, which are attractive to customers. For instance, customers can lease virtual servers in the form of access to VMs, which are guaranteed certain shares of a physical server's hardware, e.g. a specific number of cores of a multi-core CPU or a specific share of the server's main memory (RAM). Two prominent cloud computing services offering these pricing models are the Amazon Elastic Compute Cloud (EC2) and Microsoft Azure.

However, this multi-tenancy model can only work if strong inter-VM isolation in two dimensions is in place. First of all, VMs must be isolated in the spatial domain. On consolidated servers, VMs from unknown origins execute concurrently, which means that they cannot trust each other. Even the cloud computing provider cannot trust the VMs it is hosting, because the customers are free to execute whatever software they want inside their VMs. Hence, it must be guaranteed that potentially malicious VMs cannot (1) spy on memory that belongs to another customer's VM or the host and (2) it is impossible for VMs to overwrite foreign memory locations in order to provoke crashes of other VMs or even the server's control software that hosts the VMs. Nowadays, modern CPUs fulfill this requirement with hardware-assist for virtualization in their memory management subsystems [4, 5]. This also facilitates fault isolation, which means that failures that crash one VM do not propagate to other concurrent VMs, and the underlying hardware remains operational.

The second domain that must be covered by inter-VM isolation is the temporal domain. Temporal isolation, or **performance isolation**, has to guarantee that time-sharing of shared resources between VMs results in performance that is in accordance with the expectations of the tenants. It should be avoided that one VM executes malicious or uncooperative workloads that impair or steal performance from other concurrent VMs. If performance isolation is implemented poorly, the system becomes prone to performance interference. For instance, if a tenant leases a VM with 1 Gbit/s Ethernet

connectivity, it should not be possible for other VMs to steal or degrade bandwidth from this tenant's VM. Hence, strong performance isolation is important to cloud computing providers in order to offer good service level agreements, which in turn attract customers. In research, performance isolation has been a topic of study since the early days of virtualization [6, 7].

Although starting out as a server technology, the spatial and performance isolation properties of virtualization recently also caught attention in the embedded computing domain. Here, the technology promises to enable multi-core embedded systems that consolidate different sub-systems in isolated environments while reducing overall costs at the same time [8]. For example, in the automotive domain, Electronic Control Units (ECUs) could use virtualization to share multi-core hardware between mixed-criticality functions [9, 10, 11], e.g. infotainment and instrument cluster applications [12]. Besides costs, this approach would also reduce weight and cabling, and save installation space. The same goals could be achieved by consolidating multiple legacy functions, which historically run on their own ECUs, into a single but powerful virtualized multi-core ECU. Similar considerations are made for embedded systems in the avionics domain [13, 14, 15]. In contrast to the server and cloud computing landscape, inter-VM isolation requirements for embedded systems are even more strict due to additional real-time and safety requirements.

1.1 Problem Statement

While the problem of spatial isolation in virtualized systems can be considered as solved since the introduction of respective hardware-assist [4, 5], performance isolation is still work in progress. In contrast to spatial isolation, which describes a single problem for a single shared resource (restricting processes from accessing certain memory locations), performance isolation is a multi-dimensional problem. This is because modern multi-core systems inherently employ multiple shared resources, for example CPU cores, caches, interconnects or I/O devices. For each of these resources, there are multiple architectural mechanisms that can affect performance isolation [16], which aggravates the complexity of the overall problem. For some resources, solutions have significantly advanced in the meantime. For example, performance isolation issues regarding CPU caches have been recognized and acknowledged by hardware manufacturers, which resulted in the recent introduction of hardware-assist for CPU cache monitoring and allocation [17]. With its help, cache-related performance interference between multiple concurrent CPU processes like VMs can now be prevented.

Performance isolation for I/O devices, in contrast, is still a field of active research. Additionally, it is a particularly diverse problem, because there are multiple approaches and implementations for virtualizing I/O. With emulation and paravirtualization, two legacy, software-based approaches exist, which utilize a trusted software layer for relaying and multiplexing I/O of VMs to physical hardware. The latest generation of I/O virtualization is PCIe passthrough and Single Root I/O Virtualization (SR-IOV). It uses hardware-assist to self-virtualize I/O devices (SR-IOV) and directly connect

the virtual I/O device instances to VMs (passthrough). This removes the computational overhead of the software-based relaying and multiplexing layers in the host, which makes PCIe passthrough and SR-IOV the currently best-performing solution for I/O virtualization [18, 19, 20], and therefore superior to emulation and paravirtualization.

In literature, research on performance isolation of both software-based I/O virtualization approaches is plentiful, and can be categorized into three types. Studies of the first type collected data on performance isolation issues of certain virtualization solutions. This was achieved by running multiple diverse workloads or benchmarks in VMs and quantifying performance interference effects between them [6, 7, 21, 22]. Others focused on specific I/O resources like storage [23] or networking [24]. Studies of the second type contain approaches for improving isolation, mostly by enhancing different aspects of VM scheduling. For example, by fairly accounting for VM-demand of CPU cycles [25, 26], adapting to the communication behavior and need of VMs [27, 28, 29], or by optimizing placement of tasks or VMs in the virtualized server or datacenter [30, 31, 32]. The third type investigates means to deliberately degrade performance of concurrent VMs [33, 34].

However, until now, little work exists on performance isolation for the latest generation of I/O virtualization, PCIe passthrough and SR-IOV. Research on this topic is important, because the technology has already been deployed by cloud computing providers like Amazon, and it is also considered for future embedded systems in the automotive domain [12].

1.2 Contributions

The goals of this thesis are to (1) provide a comprehensive investigation about performance isolation issues of PCIe passthrough and SR-IOV, and to (2) introduce domain-specific solutions for cloud computing and embedded systems that mitigate and resolve performance isolation shortcomings of current implementations.

First, based on work in [35, 36], it is introduced that performance isolation of PCIe passthrough and SR-IOV can be broken with Denial-of-Service (DoS) attacks from malicious or buggy VMs, which flood passthrough and SR-IOV hardware with spurious PCIe packets. Accordingly, a threat model is formulated and a sample implementation of a DoS attack is presented. Results from four experiments on a system comprised of commercial-off-the-shelf hardware are presented. They demonstrate that latencies and throughput of different shared resources degrade significantly during DoS attacks on an SR-IOV capable 1 Gbit/s Ethernet NIC. Insights from the experiments are used to construct an abstract system model that gives a concise overview of architectural issues within PCIe passthrough and SR-IOV, which enable the DoS attack exploit. Furthermore, a classification of DoS attacks into four types is presented, which can be used in future work to communicate the impact and severeness of a specific DoS attack.

Second, based on work in [37], a lightweight solution that mitigates performance isolation attacks in cloud computing systems is proposed. The decision to follow a mitigation approach is motivated with domain-specific requirements. A design is presented that extends current cloud computing systems with lightweight hardware monitoring

that detects DoS attacks in a live system and reports attacker VMs to the host. The latter invokes scheduling extensions that mitigate the performance isolation attack. A successful prototype implementation of this design using an SR-IOV capable FPGA development board, which is installed together with the proposed software extensions on real hardware, proves the feasibility of the approach. Finally, results of an evaluation of the prototype are presented. The evaluation was conducted using three network-based cloud computing benchmarks.

Third, based on work in [38], solutions for completely preventing DoS attacks are proposed. They are intended for use in embedded systems with strong isolation requirements that cannot be fulfilled by software-based mitigation. Therefore, two integrated hardware architectures for multi-core platforms are proposed. They are optimized for different goals; Scheduling freedom or minimal hardware costs. To achieve its specific goal, each architecture utilizes a different subset of the optional Quality-of-Service (QoS) extensions of the PCIe specification. It is determined which extensions are needed, and how virtualized multi-core CPUs have to interface and implement them. The latter aspect is explicitly not covered in the PCIe specification. An evaluation of both architectures, conducted on a SystemC model of a real-world system with SR-IOV hardware, shows that the proposed architectures successfully close the DoS attack vector. The results are also used to compare and discuss the pros and cons of each individual approach.

The contributions and results of this thesis were published at three international conferences [35, 37, 38] and in one journal article [36]. To the best of my knowledge, the presented work is the first that comprehensively investigates DoS attacks on PCIe passthrough and SR-IOV, and contributes towards solving the resulting performance isolation issues. A graphical representation of the contributions is depicted in Figure 1.1.

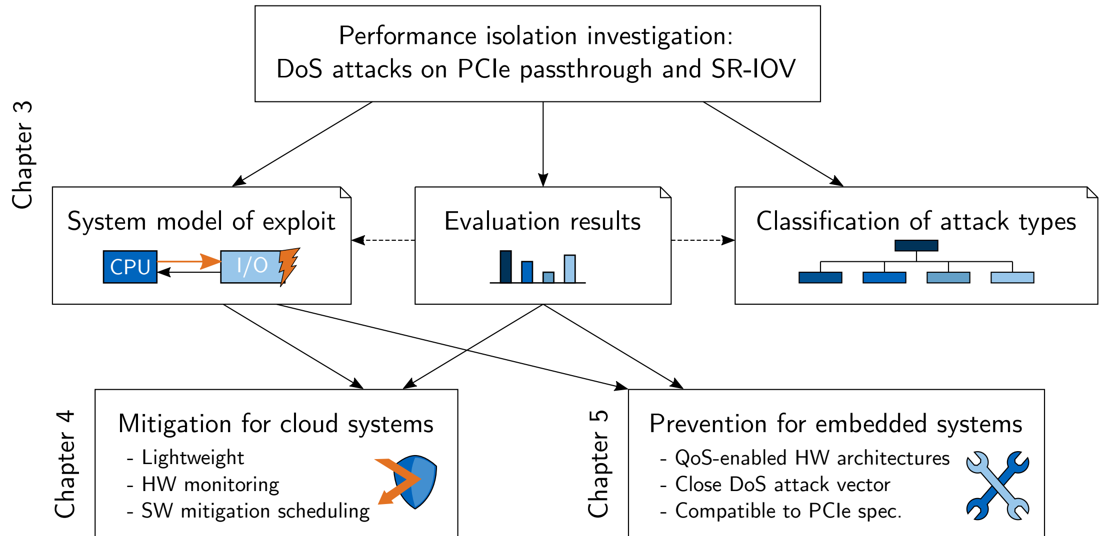


Figure 1.1: Contributions to the field of performance isolation for virtualized multi-core systems using PCIe passthrough and SR-IOV.

1.3 Outline

The remaining part of this thesis is structured as follows: Chapter 2 provides state-of-the-art on relevant technologies and computing domains. The investigation of performance isolation issues of PCIe passthrough and SR-IOV is presented in Chapter 3. Chapter 4 introduces the lightweight hardware/software approach for mitigating performance isolation attacks in cloud computing systems. Performance isolation solutions for virtualized embedded systems are presented in Chapter 5. Finally, Chapter 6 concludes this thesis by comparing and discussing all presented solutions and giving an outlook on future work on this topic.

2 State of the Art

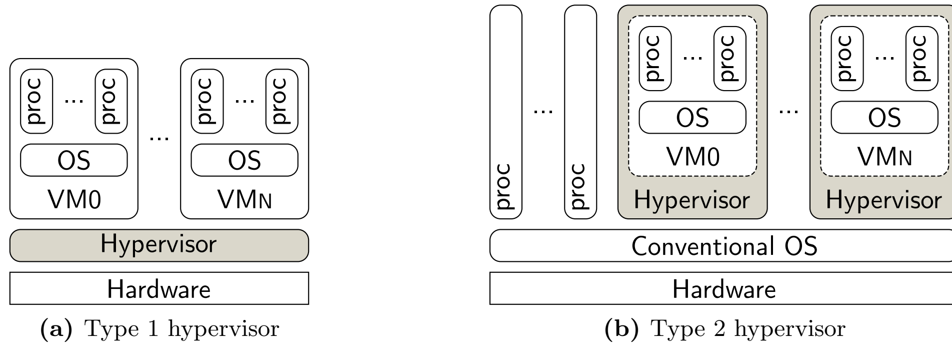
This chapter introduces the state of the art in platform virtualization and I/O performance isolation. It is structured in a top-down fashion. First, Section 2.1 presents how platform virtualization in general enables sharing of modern multi-core hardware between multiple Virtual Machines. Next, spatial isolation and performance isolation are introduced, and the computing domains that have adopted virtualization technology are presented. Afterwards, virtualization technologies for I/O are introduced in Sections 2.2 and 2.3, together with current approaches and research results that investigate and/or improve their performance isolation. The I/O virtualization technologies are presented in the chronological order they surfaced: Emulation, paravirtualization and passthrough I/O with Single Root I/O Virtualization (SR-IOV).

As this thesis contributes to the latter approach (passthrough I/O), it is the most extensively covered part in this chapter. At the time of writing, commercial off-the-shelf (COTS) passthrough I/O devices are mainly provided for the PCIe interconnect. This is a result of x86, which traditionally employs PCIe, being the dominant processor architecture in cloud computing, a domain in which platform virtualization was a key enabling technology [1]. For these reasons, this thesis primarily focuses the x86 family of processor architectures and PCIe-based I/O devices and virtualization technology.

2.1 Platform Virtualization

Platform virtualization enables the generation of simulated environments of real computer hardware. These environments are called Virtual Machines (VMs). VMs can be dynamically created and destroyed during runtime by a trusted software layer that is usually called hypervisor, or simply the host (both terms are used interchangeably in the following). Correspondingly, VMs may also be called guests or guest software. Inside VMs, it is possible to execute any kind of software, including whole Operating Systems (OSs), and it is possible to host multiple VMs on the same physical hardware. Virtualization also offers fault isolation: Critical faults inside VMs only crash the software that is running encapsulated inside the faulting VM; Other VMs or the host are not affected. The concept of virtualization has been around since the 1970s. At the beginning, hypervisors were categorized into two types [39]. Type 1 hypervisors (bare metal) run directly on physical hardware, while type 2 hypervisors (hosted) execute as userspace processes within a conventional OS. A block diagram of both types is depicted in Figure 2.1.

In the past, type 1 hypervisors had better overall performance because the hypervisor software layer executes in kernel mode and could be specifically designed to multiplex hardware access between multiple VMs with low overhead. Xen [40] is an example of a



(a) Type 1 hypervisor

(b) Type 2 hypervisor

Figure 2.1: Comparison of classic hypervisor types.

type 1 hypervisor. In contrast, type 2 hypervisors needed to execute hardware multiplexing through the interfaces of a conventional host OS. These interfaces were not optimized for such tasks, which resulted in more overhead and therefore less VM performance [41]. VMware Workstation [42] is an example of a type 2 hypervisor. Nowadays, however, the picture has changed. Since the advent of hardware assist for virtualization [2, 3], it is no longer possible to clearly label hypervisors as type 1 or type 2. For example, KVM-accelerated QEMU [43, 44] on Linux consists of a userspace process (QEMU) and a kernel module (KVM). This combination shows traits of both hypervisor types. On the one hand, QEMU is hosted by the Linux kernel (type 2), on the other hand, it also uses the KVM module that effectively converts Linux to a type 1 hypervisor. Furthermore, it is no longer possible to infer performance differences between hypervisors solely from their type. It is rather the case that overhead incurred by different hypervisors varies depending on application type and used resources [45].

2.1.1 Spatial Isolation

Spatial isolation must ensure that it is not possible for one VM to willingly or unwillingly read or alter any kind of memory resource that is assigned to another VM or the host. This affects the system's main memory (RAM) and I/O device memory such as configuration registers. This must be ensured for both, direct (CPU cores) and indirect (DMA) memory transactions. In other words, it is a security requirement that has to prevent malicious VMs from executing CPU instructions that read foreign memory that belongs to concurrent VMs or the host. The same is true for write instructions that could corrupt foreign memory and potentially crash concurrent VMs or the whole host system. Fortunately, this kind of isolation is inherently provided by the technologies that enable platform virtualization in the first place: In order to run multiple VMs on the same physical machine, it is necessary that the hypervisor partitions the machine's memory and exposes only a private and exclusive share to each VM. This is realized by virtualizing the platform's MMU subsystems. In the past, a software emulation technique called shadow page tables [46] was used. Nowadays, there is hardware-assist for virtualization in most x86 MMUs [4, 5], which is significantly faster than employing shadow

page tables in most cases [47]. Virtualized MMUs in recent server CPUs using modern hypervisors almost reach bare metal performance, incurring less than 5% overhead [48]. Hardware-assist for I/O virtualization has also been introduced for IOMMUs [49, 50]. These VM-isolation properties made virtualization "the primary security mechanism in today's clouds", according to [51].

2.1.2 Performance Isolation

Temporal isolation, also called performance isolation, describes the capability of a host system to prevent or limit performance interference amongst VMs. It is naturally caused by time-sharing of shared resources between multiple VMs. If the sharing implementation of the host lacks proper isolation or partitioning capabilities, it is possible that the performance of one VM degrades due to uncooperative or malicious usage of shared resources in another VM. A prominent example is sharing the last level cache (LLC) of multi-core CPUs. If two VMs concurrently run memory intensive workloads, data from one VM is constantly evicted by the other, and vice versa, which results in performance interference [52, 53, 54, 55]. The problem has been solved recently by integrating hardware-assist for cache monitoring and partitioning into CPUs [17]. Similar interference effects can surface for virtualized I/O. Here, the state-of-the-art in performance isolation will be reviewed in detail in the following Sections 2.2 and 2.3.

2.1.3 Cloud Computing

Platform virtualization was, and still is, an important enablement technology for Cloud Computing [1], especially, but not exclusively, for the Infrastructure-as-a-Service (IaaS) service model. This becomes apparent from The National Institute of Standards and Technology (NIST) definition of cloud computing [56], which defines IaaS as follows:

> " The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; [...] "

Additionally, the NIST definition demands that provisioning must feature dynamic and on-demand characteristics. All of this can be conveniently realized with virtualization. Hypervisors of the cloud computing providers can dynamically generate VMs that provide the consumer's requested hardware resources (number of CPU cores, amount of storage, etc.), and consumer software can then be executed inside the VMs.

This model is also beneficial to the cloud computing provider itself. It enables multi-tenancy by consolidating multiple VMs onto the same physical machine through partitioning and sharing of hardware resources. This results in high utilization of hardware, which significantly improves efficiency. Additionally, the option to perform VM live migration [57] facilitates fault management, load balancing and hardware maintenance, while downtimes for customers are reduced.

2.1.4 Embedded Virtualization

Platform virtualization technology is also seeing more and more adoption in the embedded computing domain. In [8], Gernot Heiser argues that the technology will help to enable future embedded systems where different sub-systems can co-exist in isolated environments, while overall system costs are reduced at the same time. He presents three use cases for embedded virtualization: (1) Consumer-electronics devices, like smartphones, where real-time environments (e.g. baseband software) are co-located with rich OSs (e.g. Android or iOS). (2) Devices with isolated environments for safety- or security-critical components, e.g. secure communication devices or medical devices. (3) Cars, where infotainment software and automotive control and convenience functionality is consolidated on a single Electronic Control Unit (ECU).

In fact, a virtualized phone has already been launched by Motorola [58]. Frameworks for smartphone virtualization were also presented by VMware [59] and Lang et al. [60]. In the automotive domain, recent research suggests that the isolation properties of virtualization are a reasonable means to exploit the power of multi-core CPUs by consolidating multiple functions of mixed-criticalities on a shared multi-core ECU [9, 10, 11]. For instance, work in [12] presents the consolidation of a virtualized general purpose Android infotainment application and a virtualized instrument cluster application on a shared x86 automotive ECU. Here, virtualization isolates potential faults from the untrusted Android application from the trusted code of the car manufacturer, which runs in the instrument cluster application. Consolidation could also be realized with legacy functions that historically run on their own ECUs. In both cases, installation space, weight and cabling are reduced, and therefore overall costs are reduced and efficiency is increased at the same time. In [61], additional benefits of virtualization for automotive embedded systems, besides isolated consolidation, are presented. In the avionics domain, virtualization is seen as a suitable approach to implement the spatial and temporal isolation requirements of the ARINC 653 standard. Virtualization based implementations are presented in [13, 14, 15].

In contrast to hypervisors that are employed in the server and cloud computing domain, additional real-time and safety requirements are in place for embedded hypervisors, which motivates the development of more specialized or extended solutions. A comprehensive and detailed collection on the state of the art of real-time issues in embedded virtualization is presented by Gu et al. in [62]. The paper contains information on the availability of commercial hypervisors for hard real-time virtualization in safety-critical systems, e.g. OpenSynergy Coqos[1] or SYSGO PikeOS[2], as well as work that improves the real-time behavior of general purpose hypervisors like Xen and KVM, e.g. RT-Xen [63]. As this thesis has a clear focus on I/O, techniques for virtualizing I/O devices and related work on enforcing performance isolation for virtualized I/O will be presented in the following sections.

[1] http://www.opensynergy.com/en/products/coqos/
[2] https://www.sysgo.com/products/pikeos-hypervisor/

2.2 Software-based I/O Virtualization

All software-based I/O virtualization techniques share the common characteristic that I/O of VMs is, in one form or another, relayed via trusted host software. This software executes the requested I/O operations on the physical hardware on behalf of the VMs, and returns eventual results to the respective requesters. Due to the relay, the host can ensure that spatial isolation is enforced, because it can check and verify each VM access. The host is also responsible for multiplexing requests from multiple VMs if a device is to be shared. In the following, the two concepts of emulation and paravirtualization for I/O are presented.

2.2.1 Emulation

Using emulation, the host is able to create a VM that emulates the existence of a certain I/O device. Generally, it is beneficial to emulate a device that is wide-spread and popular in the real-world, so that there is a high chance for out-of-the-box support by many OSs that come into question for running as guests inside VMs. Typical examples for emulated devices are the popular Intel e1000 or the Realtek RTL8139 family of Ethernet adapters. If the guest OS is then executed inside the VM, no further modifications are needed to enable I/O, because it already possesses a driver for this emulated version of a physical I/O device.

The host uses a trap and emulate mechanism to catch any VM interactions with the emulated device. It then executes the wanted interaction on behalf of the VM on its physical hardware, and finally returns control to the VM, also emulating potential return values. If multiple VMs are running on the host, a software bridge can be interposed between the emulation layer and the host device driver, so that host hardware can be shared. I/O device emulation can be implemented in various ways, resulting in different trade-offs. In the following, three popular implementations and their pros and cons are presented. First, emulation inside a type 1 hypervisor is presented. A corresponding block diagram is depicted in Figure 2.2.

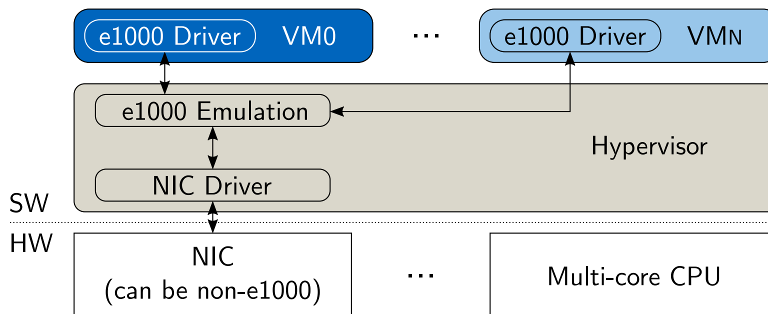


Figure 2.2: I/O device emulation in a type 1 hypervisor.

Using this design, both the emulation layer and the device driver for the physical hardware reside in the hypervisor. VMware ESX [64], for example, employed this design.

It facilitates good performance, because it requires only a small amount of VM exits and entries (CPU context switches between VM and hypervisor contexts), which are the main source of overhead in platform virtualization [65] (exits due to I/O are particularly expensive [66]). On the downside, the design defeats the paradigm of having a small trusted computing base (TCB) in order to satisfy security requirements. Here, the hypervisor has to include code for emulation and device drivers. This bloats the codebase of the trusted part of the system, which increases the probability of bugs, which in turn enlarges the attack surface. Additionally, support for hardware is constraint by the availability of respective device drivers in the hypervisor. These problems can be addressed by employing a trusted VM, like depicted in Figure 2.3.

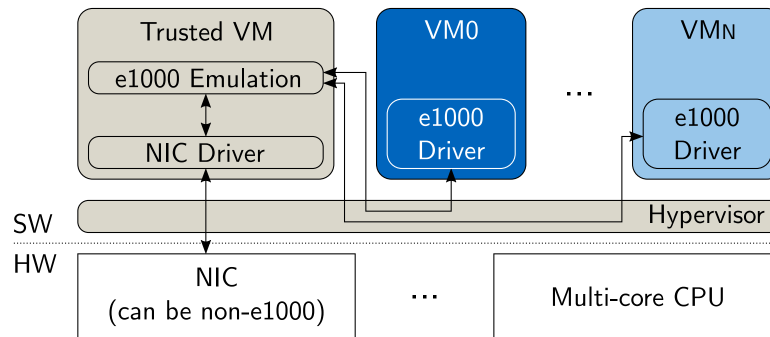


Figure 2.3: I/O device emulation with type 1 hypervisor and trusted VM.

The trusted VM is intended to assist the hypervisor. It usually runs a modified version of a conventional OS like Linux, which provides broader driver support than a specialized hypervisor. Xen [40] is a typical example that uses a trusted VM[3]. Physical I/O devices are directly exposed to and handled by drivers in the trusted VM, which therefore increases hardware support and slims down the codebase of the hypervisor. Further slim-down is realized by also offloading the emulation layer to the trusted VM. In this design, the lean hypervisor is still trapping VM access to emulated I/O devices, but forwards the emulation work to the trusted VM. The drawback of this approach is more VM exits and entries, because there are additional switches between hypervisor and trusted VM, which ultimately result in less performance. A similar design is used in hypervisors that are hosted. This third design approach towards emulation is depicted in Figure 2.4.

Like in a trusted VM, emulation is also computed in a userspace process (the hypervisor process), but the actual device driver is provided by the conventional host OS. This increases hardware support even further, because the hypervisor code must only be compatible to the networking API of the host OS, which abstracts from the actual device driver. Hence, in contrast to the previous design, it must not be ensured that a trusted VM is available that provides driver support for the targeted hardware. The hypervisor can run on any hardware for which the host OS provides a driver, which also

[3]In literature, many alternative names for this VM exist. For example, in the terminology of Xen, it may also be called privileged VM, integrated driver domain (IDD) or Dom0.

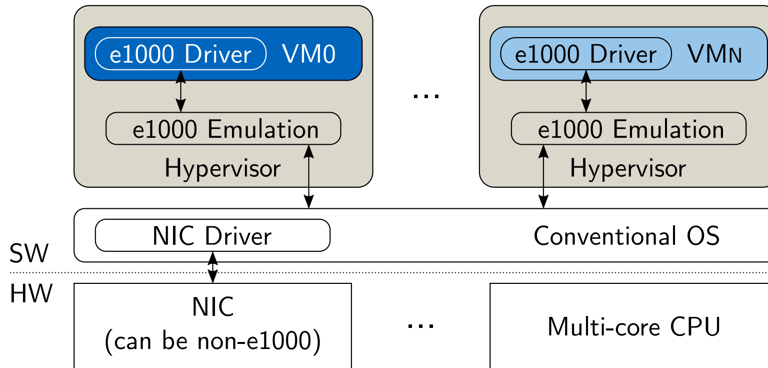


Figure 2.4: I/O device emulation in hosted hypervisors.

includes future hardware. On the downside, overall performance might be slightly worse because the hypervisor itself is hosted.

In conclusion, the choice of how to implement I/O device emulation is a trade-off between security (size of trusted code base), performance and hardware support. For sophisticated I/O devices like GPUs or network interface cards, however, all approaches have in common that emulation is inherently inefficient, because in contrast to native I/O with a single device driver, a lot of overhead is introduced. Emulated I/O must traverse the guest device driver, the emulation layer and finally the host device driver. This results in moderate performance and high CPU utilization due to the computational overhead in the emulation layers of the host. As a result, emulation is mostly employed for simple devices or peripherals, e.g. system timers.

2.2.2 Paravirtualization

The unwanted overhead of emulation can be reduced by employing paravirtualized I/O, an approach that has been popularized by the Xen hypervisor [40, 67]. Paravirtualization lets guest OSs in VMs "cooperate" with the host software. The guest loads a so-called front-end driver that communicates with a back-end driver that is operated by the host. This is also called a split-driver structure. Payload data between front- and back-end drivers is transported via shared memory, and notifications about new data are exchanged via the hypervisor. This design is compatible to each of the three virtualization implementations that were presented in the previous section. Basically, the device driver in the guest OS is replaced by the front-end driver, and the emulation layer in the host is substituted by the back-end driver. To give an example, the resulting design for a virtualized system that utilizes a trusted VM is depicted in Figure 2.5.

The directed communication between VM and host removes the trap-and-emulate overhead of emulation solutions and therefore enhances performance. On the downside, paravirtualized I/O can only be used if there is a front-end driver available for the respective guest OS that is compatible to the host's back-end driver. While this was not always the case in the early days of (para)virtualization, it is less of a hurdle nowadays. Thanks to virtio [68], a de-facto standard for paravirtualized drivers has been established

that is continuously improved and supported by most of the typical guest OSs like Windows and Linux.

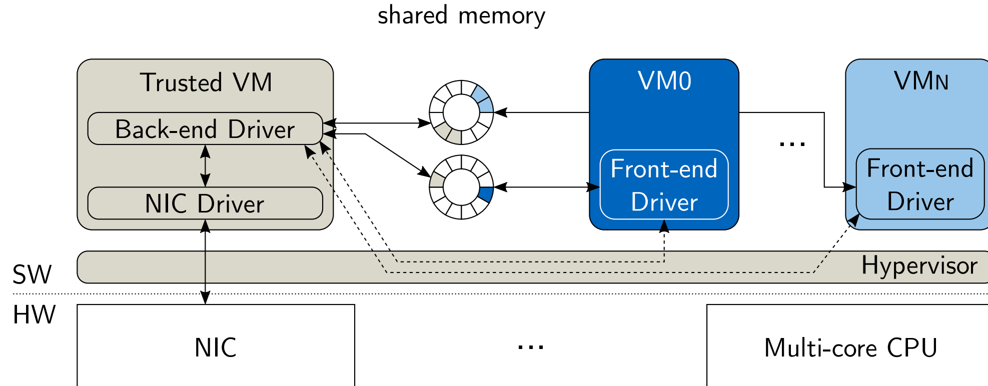


Figure 2.5: Paravirtualized I/O with a trusted VM. Front-end and back-end drivers cooperatively exchange data via shared memory. Notifications about new data are communicated via the hypervisor.

2.2.3 Performance, Pros and Cons

Like mentioned before, paravirtualization usually outperforms emulation as it incurs less computational overhead. For example, Barham et al. [40] showed that in a network TCP benchmark using an MTU of 1500 B, the original release of Xen, using its paravirtualized network drivers, was able to achieve the performance of a native Linux system (897 Mbit/s for both TX and RX). VMware workstation, using an emulated network device, achieved 68% less bandwidth for TX and 31% less for RX.

However, even paravirtualization suffers from performance degradation if platforms have sophisticated I/O configurations, despite various optimization efforts to identify [69] and reduce the computational overhead [70, 71]. For instance, if network subsystems are employed that go beyond bandwidths of 1 Gbit/s, CPU overhead is still a limiting factor. Like shown in [18], paravirtualization was not able to achieve native performance on a 10 Gbit/s Ethernet NIC in RX and TX throughput benchmarks for most of the employed message sizes. In RX tests with big message sizes, paravirtualization saturated around 6 Gbit/s compared to 9.3 Gbit/s for a non-virtualized setup. Consequently, research began shifting the focus on offloading virtualization overheads into the I/O hardware itself to alleviate the CPU overhead [72, 73, 74]. Despite further efforts to improve paravirtualization for high-performance I/O [75], commercial I/O devices with hardware support for virtualization finally surfaced. They will be covered in detail in Section 2.3.2.

An advantage of emulation and paravirtualization solutions for I/O is the abstraction from physical hardware. The state of guest OS I/O subsystems is entirely resident in memory, as there is no direct communication between guests and physical I/O devices. This enables flexible control, e.g. VMs can be started and stopped on-demand, and it facilitates live replication and migration of VMs between different physical hosts.

2.2.4 Performance Isolation Approaches

Performance isolation for software-based I/O virtualization has been an active field of research ever since. Many studies focused on investigating performance interference effects between VMs that were running multiple diverse workloads or benchmarks, and therefore covered software-based I/O virtualization at least partially. Koh et al. [6] utilized an instrumented Xen to run a range of benchmarks, i.a. disk I/O workloads, in two VMs and collected performance metrics and runtime characteristics. They used the data to clusterize applications that generate certain performance interferences and developed mathematical models that predict application performance. Matthews et al. [7] designed a performance isolation benchmark that measures the impact of misbehaving VMs on concurrent VMs. The benchmark stresses different components like CPU, memory, disk and network I/O. They conducted their benchmarks on multiple virtualization solutions that also contained Xen and VMware Workstation. Further studies that cover performance isolation of software-based I/O virtualization can be found in [21, 22, 23, 24].

Effort also went into enhancing scheduling of the host or developing VM placement strategies in order to improve performance isolation of VMs. Gupta et al. [25, 26] were among the first to investigate this topic. They extended Xen such that it was possible to measure CPU time that is spent in the host for processing I/O on behalf of a guest. This time is added to the actual CPU time that a guest is consuming. The overall time is then accounted for by a new VM scheduler, which resulted in enhanced I/O fairness and better overall performance isolation. Govindan et al. [27] developed a CPU scheduling algorithm for the host that takes the I/O behavior of VMs into account for its decision-making. Using their scheduling extensions, a streaming media server was enabled to serve 3.5 times more clients than before, and response times could be boosted by 35%. Additional work on host scheduling optimizations and VM placement is presented in [30, 28, 29, 31, 32].

There is also previous work that researched malicious workloads inside VMs, which is one of the main topics of this thesis. Malicious workloads differ from "standard" workloads in the sense that they do not necessarily try to get the best performance for themselves by disregarding others, but deliberately try to degrade the performance from other concurrent VMs without any benefits of their own. Yang et al. [33] presented a performance measurement and analysis framework for virtualized I/O. It can be used to gain insight about hypervisor I/O scheduling and leverage the information to carry out disk I/O performance degradation attacks on concurrent VMs. The attacks were successful on local testbeds as well as on Amazon EC2 cloud instances. Chiang et al. [34] demonstrated a similar attacker framework for EC2 that targets network I/O instead.

Although this thesis covers performance isolation issues, the presented approaches to increase isolation could not be transferred. This is because this thesis covers hardware- instead of software-based I/O virtualization. In hardware-based I/O virtualization, VMs directly communicate with I/O hardware, and go unnoticed by the host. Hence, there is no relaying layer in the host (emulation software or back-end driver) that can be leveraged for insight into I/O consumption, attack detection or fairer scheduling. In the following section, these differences will be introduced in detail.

2.3 Hardware-based I/O Virtualization: Passthrough I/O with PCIe and SR-IOV

Passthrough I/O, in contrast to emulation and paravirtualization, directly assigns physical I/O devices to VMs. Direct assignment means that memory regions of I/O devices and VMs are directly exposed to each other, so that both communicate in a direct manner and not via software-based emulation- or paravirtualization-interfaces. In other words, the I/O device passes through the host and directly connects to a VM. Removing the emulation- and paravirtualization software layers results in near-native I/O performance, e.g. for 1 Gbit/s Ethernet NICs [76], while significantly cutting CPU overhead at the same time. In conclusion, passthrough provides superior I/O performance with respect to emulation and paravirtualization by giving VMs direct access to dedicated physical I/O devices.

Enabling passthrough for untrusted VMs requires special hardware extensions in order to retain spatial isolation requirements of virtualized systems. Therefore, modern x86 CPUs and chipsets come equipped with virtualization enabled MMUs [4, 5] and IOMMUs (Intel VT-d [49], AMD [50]). The MMU ensures that VMs cannot use their assigned CPU cores to read and write from any memory (RAM, I/O devices, etc.) that belongs to other VMs or the host system. Likewise, the IOMMU ensures that an I/O device's DMA engine can only read from memory regions that belong to the specific VM it is assigned to. Both MMU and IOMMU are configured dynamically by trusted hosts like KVM or Xen on VM creation [77]. The concept is depicted in Figure 2.6.

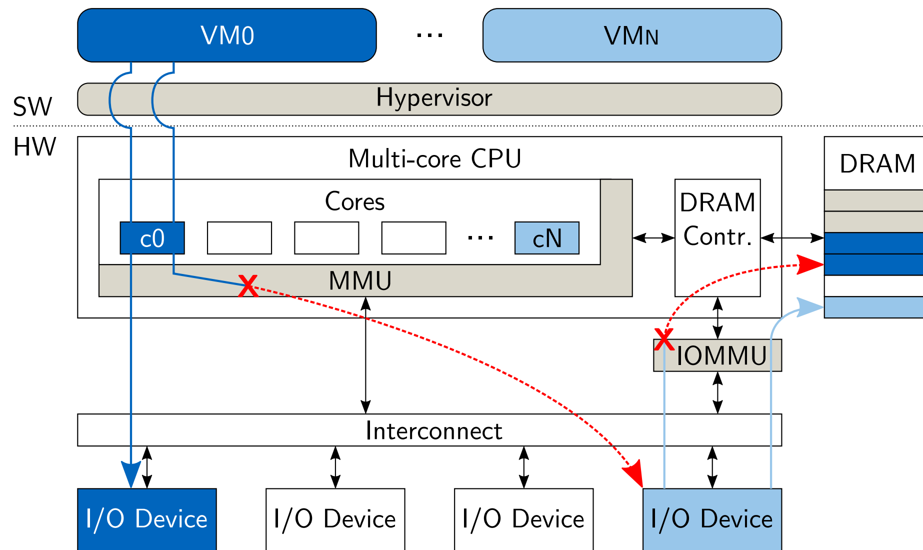


Figure 2.6: Passthrough I/O example with two VMs assigned to different I/O devices. VM0 runs on core c0 and bypasses the hypervisor for direct access to its assigned I/O device. However, VM0 is blocked by the MMU from accessing VMN's I/O device. Furthermore, the IOMMU ensures that VMN's assigned I/O device has DMA access only to memory that belongs to VMN, but not to concurrent VMs or the hypervisor.

As mentioned initially in this chapter, COTS passthrough I/O devices are mainly provided for x86 architecture platforms that utilize the PCIe interconnect. For this reason, the PCIe characteristics that are relevant to this thesis shall be briefly introduced in the following.

2.3.1 PCI Express (PCIe)

PCIe [78] is a high-speed serial interconnect that replaced the older PCI [79] and PCI-X [80] busses. The PCIe specifications comprise hundreds of pages describing the core technology, as well as optional extensions and concepts for achieving backward compatibility with legacy PCI. As a consequence, the following introduction will only cover the respective parts that are necessary for understanding the PCIe related parts of this thesis. Also, sometimes an abstracted high-level view is presented if a focus on concepts is more important than actual details of the technical specification.

2.3.1.1 Topology

Unlike PCI and PCI-X, which utilize a shared parallel bus topology, PCIe is a serial, point-to-point, packet-switched interconnect. A typical PCIe topology has a single root node, called the "Root Complex", which connects the CPU and memory (DRAM) subsystems to the PCIe I/O subsystem. The Root Complex may have multiple PCIe ports that either connect to switches or PCIe devices (also called Endpoints). Recent x86 CPUs directly integrate the Root Complex on the processor die. It is also possible to connect legacy PCI/PCI-X devices to PCIe via PCIe to PCI bridges. However, legacy PCI is not in the scope of this thesis and will therefore not be considered further. An example PCIe topology is shown in Figure 2.7.

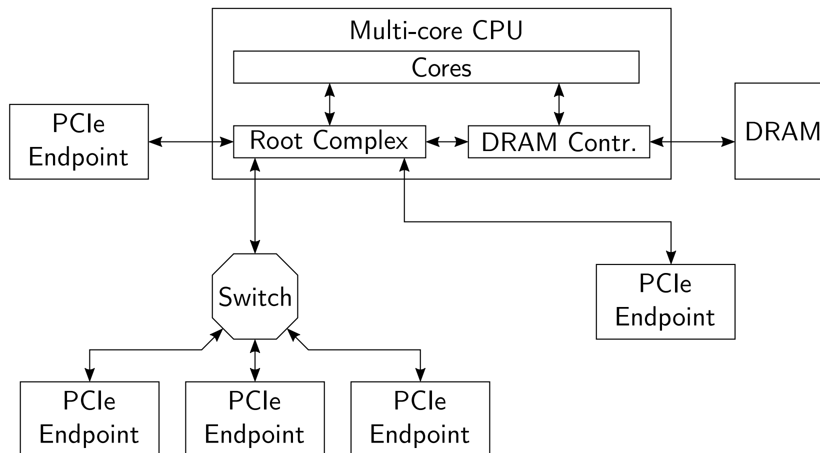


Figure 2.7: Example PCIe topology of a multi-core CPU with integrated Root Complex.

2.3.1.2 Link Parameters and Bit Rates

The communications channel between two PCIe components is called a link. A link has three parameters that define its actual net bit rate. First, each link has a number of lanes, which indicate the number of physical signal pairs between two PCIe components. The PCIe specification denotes the number of lanes by xN. For example, x16 means the link supports 16 lanes. Second, each lane has a physical layer gross bit rate (R_b), which is given in so-called GigaTransfers per second (GT/s). The supported bit rate depends on the PCIe version that the respective hardware implements. Third, each PCIe version uses a specific encoding for transfers on the physical layer. An overview of supported parameters depending on PCIe version is given in Table 2.1.

Table 2.1: Performance parameters by PCIe version.

PCIe version	R_b (GT/s)	Encoding
1.0	2.5	8b/10b
2.0	5	8b/10b
3.0	8	128b/130b

Please note that newer PCIe versions are backwards compatible in terms of speed. For example, a PCIe 3.0 slot can still operate at 2.5 GT/s with an encoding of 8b/10b if a PCIe 1.0 device is installed in it. Given all three parameters, it is possible to calculate the physical layer net bit rate (R_{net}). For instance, R_{net} of an x16 link with 8 GT/s using 128b/130b encoding is computed as follows:

$$R_{\text{net}} = 16 \cdot 8\,\frac{\text{GT}}{\text{s}} \cdot \frac{128}{130} = 126\,\frac{\text{Gbit}}{\text{s}} = 15.75\,\frac{\text{GB}}{\text{s}}.$$

2.3.1.3 PCIe Functions, Memory Mapped I/O and Drivers

Although physical PCIe components like Endpoints are connected by a single link, the standard allows to partition components into multiple so-called PCIe functions. Each individual function is independently addressable by other system components like CPU cores. Therefore, each function must first be mapped into the system's memory map, a task that is usually executed during system boot by firmware or the OS. For instance, in x86 based systems, memory mapping is done by the BIOS. Each PCIe function requests memory resources from the BIOS, which reserves a respective address range in the system's memory map. Afterwards, the BIOS reports back to each function and notifies it which address range it got. This way, it is possible for a function to determine if it is the destination of an inbound PCIe packet by comparing the packet's target address with its assigned address range.

As soon as all PCIe functions are mapped, they are ready to receive CPU-to-PCIe transactions. If the destination address of a CPU core's machine instruction translates to a PCIe function address, a respective packet is compiled and put on the PCIe interconnect. This method for performing I/O operations is called Memory Mapped I/O

(MMIO). It enables the system to consolidate both DRAM and I/O device addresses into a single memory map. As a consequence, the CPU can use the same machine instructions for both MMIO and DRAM transactions, which speeds up I/O transactions and simplifies device driver writing. PCIe devices that support Direct Memory Access (DMA) utilize the same system memory map, which means they can read from DRAM as well as other PCIe devices that are mapped via MMIO.

Common examples for real-world multi-function PCIe devices are multi-port NICs, e.g. Ethernet or InfiniBand NICs. Such a NIC's function address range is used by system software to interface to the hardware resources needed for operating a single physical NIC port. Usually, it is an OS device driver that interfaces to PCIe functions. The device driver first requests the function's base address, aka the first address of the function in the system memory map. Afterwards, it can access different resources of the function (e.g. registers, FIFOs, etc.) by adding an offset to the base address. A per-function assignment of offsets to resources is known to the programmer of the device driver, usually by consulting the device's datasheet. Most of the time, device drivers are written for single PCIe functions, not for the whole device. For example, if a device provides multiple identical functions, Linux loads multiple instances of the same driver, but supplies different base addresses (aka PCIe functions) to each driver instance.

The PCIe specification allows for up to eight functions per PCIe device, except for devices supporting an optional PCIe capability called Alternative Routing-ID Interpreation (ARI), which adds support for up to 256 functions. Of course it is in the responsibility of the PCIe device designers and engineers to choose link parameters that satisfy the combined net bit rate demands of all provided PCIe functions (see Section 2.3.1.2). A summary of this section using the example of a multi-port Ethernet NIC is depicted in Figure 2.8.

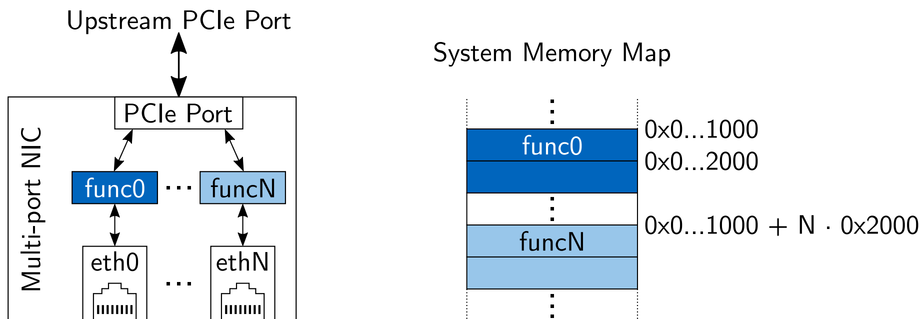


Figure 2.8: A multi-function PCIe Endpoint using the example of a multi-port Ethernet NIC. Each device function requests 8 KiB of memory and is assigned a respective address range in the system's memory map.

Thanks to the representation of PCIe functions as address ranges in the system memory map, it is possible to do passthrough I/O at function granularity. The host realizes this by directly mapping PCIe function address ranges into a VM's virtual address space.

2.3.1.4 Transaction Characteristics and Flow Control

As mentioned in 2.3.1, single PCIe transactions are sent via packets. Read transactions are so called non-posted requests, which means that the reader has to wait for a completion packet from the target that contains the requested data word. For instance, a CPU core would block on the machine instruction that reads from a PCIe device until the completion is received. In contrast, write transactions are classified as posted requests. Here, program execution can continue as soon as the write packet has been sent to its first downstream stop; write packets are not completed, aka acknowledged, by the receiver.

PCIe also incorporates hardware flow control mechanisms, which ensure that PCIe packets are sent out only if a free downstream buffer slot is available. For instance, if a CPU core wants to execute a write instruction to PCIe memory, but backpressure from flow control signals zero available buffers, the core would block on the write instruction until a downstream buffer slot is freed.

2.3.2 PCIe Single Root I/O Virtulization (SR-IOV)

While passthrough I/O offers near-native performance, it has limitations in terms of scalability. Even on machines with moderate VM count, hardware costs quickly get expensive if one full-blown passthrough device per VM is needed. However, in contrast to emulation or paravirtualization, there is no possibility to share a physical I/O device between multiple VMs with passthrough. This is because a trusted software intermediary like a hypervisor, that would be able to multiplex single I/O devices between multiple VMs, has deliberately been removed in the first place in order to enable the performance benefits of passthrough.

For combining both traits, (1) the performance of direct access via passthrough and (2) sharing a device's physical function, research has suggested self-virtualizing I/O devices [73, 74]. The idea was to overcome the performance penalties of software-based I/O virtualization and sharing routines [69] by offloading them into the I/O device hardware. This way, self-virtualizing devices can offer multiple virtual interfaces per physical device function, which are enabled by hardware-accelerated I/O virtualization and sharing.

In line with these ideas, the Single Root I/O Virtualization and Sharing Specification (SR-IOV) [81] was released by the PCI Special Interest Group (PCI-SIG). It is specified for PCIe topologies that utilize a single Root Complex. There is also MR-IOV, a specification for topologies with multiple Root Complexes [82]. However, this thesis will not specifically cover MR-IOV, because the technology is not widely used yet and the topics addressed in this thesis are mostly agnostic of the topology being single or multi-root.

2.3.2.1 Physical and Virtual Functions

Devices supporting SR-IOV provide at least one PCIe function (see 2.3.1.3) that supports the SR-IOV capability. Such functions are called Physical Functions (PFs) and they are intended to be controlled by the trusted host. Through the PF interface, it is possible

to dynamically spawn multiple “light-weight” PCIe functions, called Virtual Functions (VFs) in SR-IOV terms. VFs share physical resources with their co-created VFs and the PF. The maximum number of supported VFs per PF depends on the capabilities and hardware implementation of the actual PCIe device.

From the system's point of view, there is no difference between classic PCIe functions, PFs and VFs. Just like classic functions, there is a reserved address range in the system's memory map for each VF (compare Figure 2.8). Hence, VFs are automatically compatible with PCIe passthrough and are therefore protected by hardware-enforced spatial isolation via MMU and IOMMU. Manufacturers or vendors of SR-IOV devices usually supply special VF drivers for usage in VMs. VF drivers are more light-weight than classic drivers, because they can spare most control plane functionality for administering the physical hardware, a task which is reserved to host and its PF driver. The host also decides which VMs share physical hardware by attaching VMs to VFs that are associated with the same PF. A summary of the concepts described in this section is depicted in the block diagram in Figure 2.9.

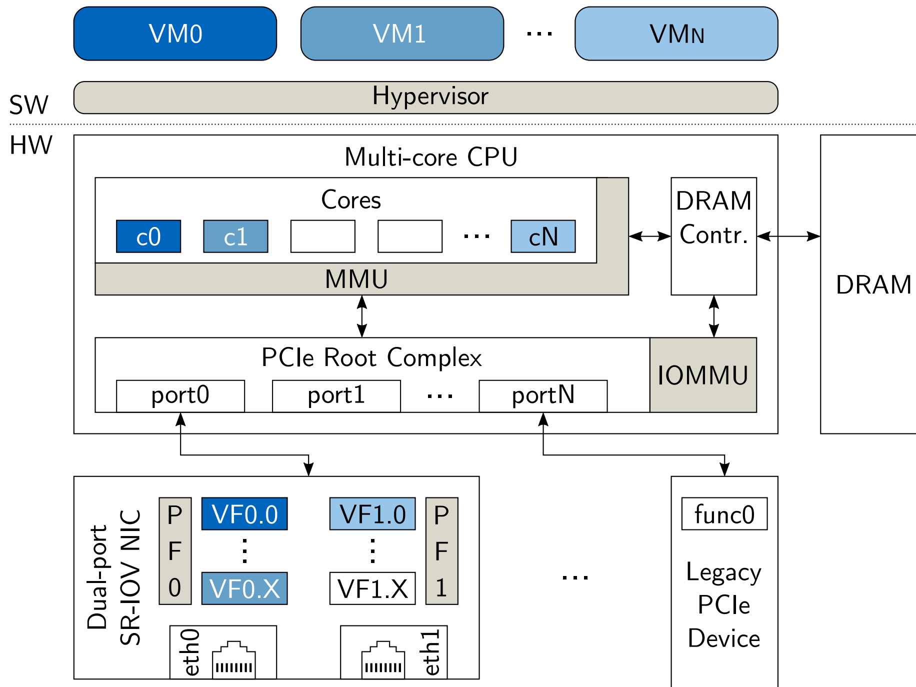


Figure 2.9: SR-IOV enables virtualization and sharing of physical resources like Ethernet ports between multiple VMs. In this example, VM0 and VM1 share NIC port eth0.

This example shows an SR-IOV capable dual-port Ethernet NIC. The physical ports eth0 and eth1 are associated with PF0 and PF1, respectively. They are administered by the hypervisor and used to spawn up to X VFs per PF. VM0, which runs on core c0, has passthrough access to VF0.0, the first VF of PF0. VM1, which runs on a separate core, is attached to VF0.X, which means that VM1 shares Ethernet port eth0 with VM0. For instance, if VM0 and VM1 would both execute the same networking benchmark with the

same parameters at the same time, then each VM would achieve 50% of eth0's maximum performance, respectively. VMN, in contrast, is attached to VF1.0, which is associated with a different PF. It is therefore not affected by the network usage of concurrent VM0 or VM1.

2.3.2.2 Adoption

Being an extension to PCIe, the prevalent interconnect in x86 systems, SR-IOV has been endorsed quickly by hardware manufacturers. Thus, commercial off-the-shelf (COTS) devices supporting the technology were available after only a short period of time. At the time of writing, Ethernet adapters made up a large part of available SR-IOV hardware. However, other I/O classes or subclasses are catching up recently: In [83], an InfiniBand adapter supporting SR-IOV is evaluated. Recent revisions of the NVM Express specification [84], which specifies a PCIe based interface to non-volatile storage like Solid-State Drives (SSDs), includes support for SR-IOV; Implementation details are presented in [85]. Additionally, the first Graphics Processing Unit (GPU) with SR-IOV support has been announced recently [86].

SR-IOV hardware has already been adopted by cloud computing providers. For example, Amazon offers optional SR-IOV networking for their Elastic Compute Cloud (EC2) instances, where it is termed as “enhanced networking”. On Amazon's website, SR-IOV is advertised for providing “higher bandwidth, higher packet per second (PPS) performance, and consistently lower inter-instance latencies” [87]. The Oracle Exalogic Elastic Cloud incorporates SR-IOV capable InfiniBand adapters, which can be shared by up to 63 VMs [88]. Additionally, SR-IOV is considered as an enabler for virtualization in High Performance Computing (HPC) [89, 90].

Recently, SR-IOV also started getting attention in the embedded virtualization domain. Herber et al. [91] presented a self-virtualizing CAN controller based on SR-IOV. It is intended for use in future automotive ECUs, which consolidate the software of multiple legacy single-core ECUs on a single multi-core ECU using virtualization. In [92, 93, 94], concepts are presented that leverage SR-IOV's partitioning and isolation capabilities to share and separate access to FPGA-based reconfigurable coprocessors in heterogeneous mixed-criticality multi-core systems. Using SR-IOV for the coprocessors also provides lower overhead than software based virtualization solutions [95]. Both concepts, SR-IOV enabled CAN controllers and coprocessors, were also consolidated into a virtualized automotive ECU demonstrator platform [12]. The ECU was running a general purpose Android infotainment application and an instrument cluster application, with both having access to SR-IOV enabled CAN and coprocessors.

For the avionics domain, Münch et al. [96] evaluated SR-IOV for mixed-criticality real-time systems, and concluded that SR-IOV is a promising approach for future systems if certain criteria are met. Their utilized evaluation platform, a Freescale P4080, however, did not meet the criteria to distinguish PCIe devices at function granularity, but the problem can be solved by an approach that is presented [97].

2.3.3 Performance, Pros and Cons

Studies on 10 Gbit/s SR-IOV Ethernet NICs show that the technology enables near native I/O performance for VMs [18, 19, 20]; SR-IOV is superior to emulation and paravirtualization approaches with respect to performance and CPU overhead in both, KVM [18, 20] and Xen [19] environments. Additional research on SR-IOV demonstrates that performance can be brought even closer to native by tweaking interrupt handling [98, 99, 100] or optimizing the VF driver for multi-core [99].

Depending on the domain in which SR-IOV based systems are deployed, fast and low-overhead migration/replication of VMs might be an additional requirement that is more (cloud computing) or less (embedded systems) important. Here, SR-IOV is not as flexible as software based legacy I/O virtualization solutions, because the state of a VM depends not only on its software image in memory, but also on the state of its passthrough hardware. Here, the problem is that passthrough hardware cannot yet be stopped or frozen like the execution of VMs on a CPU, which aggravates migration. It may be further complicated if the migration target machine has different passthrough hardware. Techniques for improving this situation are proposed in [101, 102] (hardware based) and [103, 104, 105, 106] (software based), but not yet implemented in vendor-supplied drivers or COTS hardware.

2.3.4 Performance Isolation Approaches

Performance isolation for passthrough I/O and SR-IOV is not yet as broadly covered in research as it is for software-based I/O virtualization approaches. First of all, it is important to understand that passthrough I/O exposes system resources to VMs at a lower abstraction level than software-based solutions. In contrast to the latter, passthrough grants direct access to the system's PCIe interconnect, which uses a packet-based protocol. On top of the low-level PCIe protocol, VMs and passthrough devices communicate over a higher-level protocol in order to manage reception and sendout of the device's actual I/O functionality (e.g. Ethernet). This can be roughly compared to using the TCP protocol (higher level) over IP (lower level). The higher-level protocol in passthrough is communicated between the guest VM's device driver for the passthrough device and the passthrough device's controller (further details are presented in Section 3.6). In software-based solutions, the interconnect part is abstracted through the emulation or paravirtualization layers, hence full control over the interconnect remains at the trusted host software.

Consequently, in contrast to software-based I/O virtualization, passthrough solutions must additionally provide isolation for the interconnect and not only for the actual I/O functionality. This also means that there is a one-way dependency between both. If performance isolation on the interconnect cannot be guaranteed, it may be possible that performance isolation for the higher-level I/O function, like Ethernet, can be broken too. On the other hand, it may be possible to exploit the higher-level protocol for unfair Ethernet sharing, but isolation on the interconnect stays uncompromised. This dependency is depicted in the block diagram in Figure 2.10.

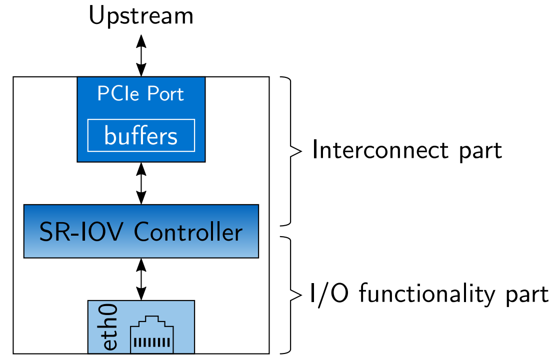


Figure 2.10: Dependency between interconnect and actual I/O functionality. If performance isolation on the interconnect cannot be guaranteed, it may be possible that isolation for the higher level I/O function, like Ethernet, can be broken too.

In one of the first papers about an SR-IOV capable COTS Ethernet NIC, Dong et al. [107] listed an outlook of possible attack types on VFs by malicious VMs, among them Denial-of-Service attacks with spurious messages. However, they only mentioned their theoretical possibility. They did not explicitly define what kind of messages would have to be used. They provided first thoughts of how countermeasures could be designed, but no further insights like an actual architecture or evaluation on real hardware were provided. In this thesis, DoS attacks are for the first time implemented, thoroughly investigated on COTS SR-IOV hardware, evaluated, and mitigation and protection approaches are presented.

For their actual I/O functionality, controllers in COTS SR-IOV devices usually implement QoS features that that enforce performance isolation. For instance, the 82576 [108] and i350 [109] 1 Gbit/s SR-IOV capable Ethernet controllers provide QoS for servicing the TX path of their Ethernet ports. Multiple VMs are either fairly arbitrated using a (weighted) round-robin scheme, or bandwidth can be partitioned arbitrarily between different VMs through the PF driver of the host. Similar functionality was presented for the self-virtualized SR-IOV CAN controller of Herber et al. [110]. They use a time-based weighted round-robin scheme for arbitration between different virtual CAN controllers. Work in this thesis instead focuses on performance isolation for the interconnect part. The results can be combined with the before mentioned approaches to provide pervasive performance isolation for both the interconnect and the actual I/O functionality of passthrough I/O devices.

Previous work on performance isolation for cache and memory subsystems focused on utilizing CPU-resident hardware performance counters to monitor access of individual cores or processes to the shared resources. On misuse or overextension, specifically designed scheduling mechanisms are responsible for enforcing isolation. Research was conducted for systems of the cloud computing [52, 53, 54] and real-time [111, 112] domain. Work presented in this thesis investigated if this approach is transferable to performance isolation for passthrough I/O and SR-IOV. The thesis will conclude that scheduling can be used to mitigate DoS attacks on passthrough I/O and SR-IOV. However, counters in

current CPUs lack needed granularity. They must therefore either be extended accordingly, or future passthrough I/O devices must employ respective monitoring facilities in the I/O devices themselves.

In [113], Bettie et al. present FPGA-based hardware real-time bridges, which are interposed between the host system's CPU and the I/O device. Performance isolation is enforced by a real-time scheduler that is connected to the bridges. The approach could be compatible to SR-IOV, but it was not implemented with support for it. In contrast, work in this thesis presents solutions that can be integrated into existing CPUs and I/O devices, enabling solutions that do not need physical hardware that is interposed.

Münch et al. [114] developed a performance isolation concept for mixed-criticality embedded real-time systems in avionics that employ SR-IOV devices. The main idea is to block CPU cores from sending any MMIO transactions at all to the I/O devices. Instead, communication with the host is realized solely through DMA transactions that are initiated by the trusted SR-IOV device. DMA transactions for the VFs are arbitrated by a static arbitration table, which enables performance isolation. This concept needs purpose-built SR-IOV devices and specific PF and VF drivers and software that is compatible to the DMA-only communication model. Solutions presented in this thesis are compatible to existing COTS device drivers and programming models that utilize host-initiated MMIO transactions.

3 Investigation of PCIe Passthrough and SR-IOV Performance Isolation Issues

This chapter presents and investigates a major finding of the thesis: Denial-of-Service attacks on PCIe passthrough and SR-IOV devices cause major performance isolation issues. As a result, malicious VMs or buggy device drivers are able to cause performance interference with concurrent VMs as well as the host.

The chapter starts by defining a general threat model in Section 3.1, and continues with Section 3.2, which explains how DoS attacks can be implemented. Subsequently, the test platform for investigating and evaluating the performance isolation issues is introduced in Section 3.3. Among other things, hardware and software configurations of the platform are presented. Section 3.4 continues by describing the evaluation methodology used on the test platform. Afterwards, Section 3.5 presents results of four distinct experiments that evaluate and investigate DoS attacks. Using the results of these experiments, Section 3.6 continues by presenting a system level model for PCIe DoS attacks on Ethernet devices that abstracts from specific hardware and software. The subsequent Section 3.7 then introduces a classification of different DoS attack types that were found. Finally, a summary of the whole chapter is presented in Section 3.8.

3.1 Threat Model

Software that performs DoS attacks on PCIe passthrough devices has a single requirement. It must have the respective access permissions for writing to the passthrough device's memory resources. In virtualized systems, there are multiple scenarios in which this requirement is either inherently fulfilled or can be fulfilled by force.

For instance, Infrastructure-as-a-Service (IaaS) cloud computing provides computing infrastructure to the customer, which means he has direct access to virtualized and passthrough hardware from the start. The customer must install and administer its own guest OS and hardware drivers inside the provided VM, which means he is also able to do as he wants with potential passthrough devices. Other service models in cloud computing abstract more from the underlying hardware. For example, Platform-as-a-Service (PaaS) offers computing platforms to the customer. Usually, they come in the form of user accounts for an OS that comes equipped with certain pre-installed execution environments and developer tools. Here, the customer does not have write permissions for potential passthrough hardware, but if he is of malicious nature, he could obtain elevated access by utilizing privilege escalation exploits in the supplied OS. Of course, it is itself already a major problem if an attacker obtains root privileges in a PaaS environment, because it enables tampering with other customer accounts that share the

same OS. However, if the compromised OS itself runs in a VM with passthrough devices and not on bare metal, the attacker can additionally cause performance interference with concurrent VMs.

Similar scenarios are possible in embedded virtualization. For instance, future consolidated mixed-criticality systems might run VMs with access to passthrough devices (compare Section 2.3.2.2). Inside such VMs, untrusted third-party software may have direct access to passthrough devices. Other VMs may run trusted code, but expose interfaces to the outside world (e.g. in-vehicle infotainment) which can be exploited by an attacker to gain access to passthrough devices.

Finally, a DoS attack in cloud or embedded systems must not always be the result of an attacker with malicious intent. There is also a possibility that a user or third-party software loads a buggy device driver for a VM's passthrough device, which accidentally causes a DoS attack.

3.2 Implementation of DoS Attacks

DoS attacks on PCIe devices work by flooding the target with spurious PCIe packets that ultimately overstrain the processing capabilities of the attacked device. Regarding the characteristics of PCIe transactions (see Section 2.3.1.4), it is clear that write transactions are most suitable for the job. In contrast to read transactions, where every single request needs to wait for the requested data word to arrive from the target device, there are no acknowledgements for write transactions. This means that write packets can be generated at much higher frequencies than read packets.

There are multiple ways to execute a DoS attack on MMIO (see Section 2.3.1.3) capable hardware. For example, an attacker can write a tiny PCIe kernel driver for the passthrough device that includes a loop that floods the device's resources. The DoS driver for the target device can then easily be loaded instead of the standard or vendor-supplied driver. Depending on the capabilities of the OS, it is also possible to launch the attack from userspace. For instance, resources from the target device can be memory mapped into a userspace process using the Linux sysfs [115] interface for PCIe devices. A minimalistic implementation in C is shown in Listing 3.1.

```
1  #define SYSFS_PCIE_FILE "/sys/bus/pci/devices/0000:08:10.1/resource0"
2
3  int main(int argc, char *argv[])
4  {
5      int target_sysfs_fd, target_size;
6      struct stat st;
7      void *target;
8
9      /* get file descriptor for target resource */
10     target_sysfs_fd = open(SYSFS_PCIE_FILE, O_RDWR | O_SYNC);
11
12     /* get memory size of the resource */
13     stat(SYSFS_PCIE_FILE, &st);
14     target_size = st.st_size;
```

```
15
16      /* memory map the resource */
17      target = mmap(0, target_size, PROT_READ | PROT_WRITE,
18                    MAP_SHARED, target_sysfs_fd, 0);
19
20      /* the actual DoS loop */
21      for (;;)
22          *(volatile uint32_t *)target = 0xD05; /* 32 bit PCIe pkts */
23  }
```

Listing 3.1: A minimalistic C implementation of a DoS attack using the Linux sysfs interface. Error handling and include files omitted for brevity.

The targeted PCIe device is defined in line one, lines 10–18 map the device's memory resources into the address space of the process, and lines 21–22 finally flood the target device with 32 bit PCIe write packets. Error handling code and include files are omitted for brevity. The DoS attack targets the address that is stored in the pointer variable called `target`. This address points at the first address of the whole address range that is reserved for the targeted PCIe device (compare Figure 2.8). What kind of I/O-device resource actually hides behind this address must be looked up in the device's data sheet. It could be a register, FIFO memory, or something else. Targeting a different resource of the device can be achieved by adding an offset to the address stored in `target`. The relevance of the actual target resource inside the PCIe device will be discussed later in Section 3.5. Flooding with PCIe packets of other sizes is achieved by casting `target` to the respective type, e.g. `uint64_t` or `uint16_t`. In the rest of this thesis, the attacked PCIe device will interchangeably be called the DoS target or DoS victim.

3.3 Evaluation Platform

The evaluation platform is based on the x86 architecture and uses a CPU and motherboard combination that was selected for building an automotive demonstrator within the ARAMiS project [116]. The goal of the demonstrator was to show that virtualization in multi-core systems is a key-enabler for consolidating different software functions onto a single physical ECU [12]. The demonstrator also utilized SR-IOV for hardware-based virtualization of a coprocessor and a CAN controller. Thus, it is a suitable choice as a base system for evaluating performance isolation issues of SR-IOV.

Figure 3.1 depicts an overview of the evaluation platform. The CPU is an Intel Core i7-3770T (3rd generation) that runs at 2.5 GHz and has four physical cores (or eight logical cores if HyperThreading is enabled). It also has a single integrated PCIe 3.0 port, which is primarily intended to house a graphics card. The CPU is mounted on an Intel DQ77MK motherboard, which provides a Q77 Platform-Controller-Hub (PCH), aka the chipset. PCH and CPU are connected by the proprietary Direct Media Interface (DMI) 2.0, which is very similar to PCIe (serial communication, point-to-point links with multiple lanes). The DMI on this motherboard is capable of 20 Gbit/s. CPU and PCH support VT-x and VT-d hardware virtualization extensions, so that PCIe passthrough is supported. The system is equipped with 32 GB of RAM.

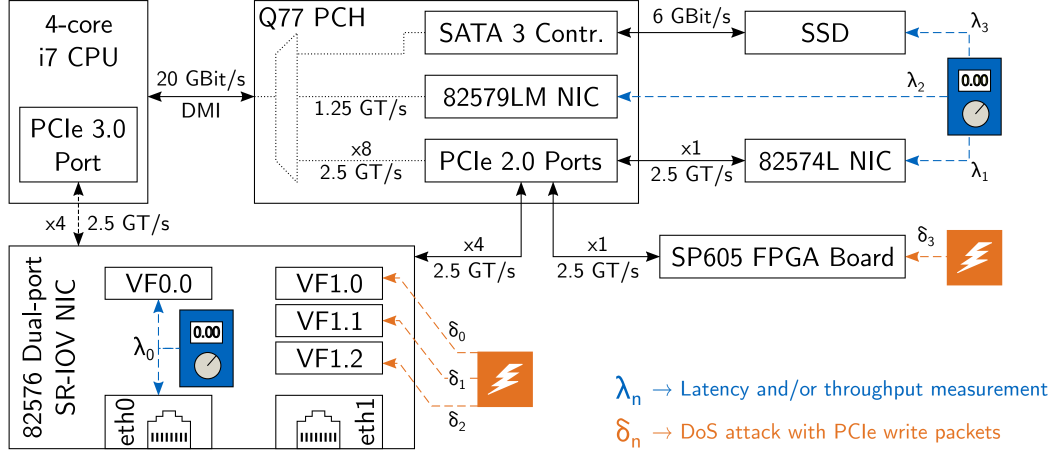


Figure 3.1: Block diagram of the evaluation platform's hardware components and their configuration. The 82576 NIC is either connected to the CPU's PCIe 3.0 port or the PCH's PCIe 2.0 port. Measurements and DoS attacks are indicated by λ_n and δ_n, respectively.

The DMI connection between CPU and PCH is transparent to software, which means that all of the PCH's PCIe ports and integrated PCIe devices appear to the OS as independent devices or ports, which are not behind a PCIe switch. It is not disclosed how multiplexing between DMI and the PCH subsystems is implemented in reality, and therefore depicted with dotted lines.

For mass storage purposes, the PCH integrates a SATA 3 controller. It has a total of six ports, with two high-speed ports that support 6 Gbit/s. Each of the two is connected to an 80 GB Solid-State Drive (SSD). While one SSD is utilized to store the system and VM images, the second is used to benchmark data throughput. Only the latter is depicted in Figure 3.1.

The PCH provides three PCIe 2.0 ports, one with four lanes (x4) and two with one lane (x1). All lanes support a bit rate of 2.5 GT/s. One of the x1 PCIe ports is hardwired to an Intel 82574L Gigabit Ethernet NIC, which is soldered directly onto the motherboard. The second x1 port is equipped with a Xilinx SP605, a PCIe development board with a Spartan 6 Field-Programmable Gate Array (FPGA). The board will be utilized as a DoS target with run-time configurable processing times for PCIe packets. Its purpose will be discussed in detail in Section 3.5.2. A second Gigabit Ethernet NIC, an Intel 82579LM, is directly integrated into the PCH. To software, the NIC looks like a standard PCIe device so that the default PCIe drivers for the controller can be used. In terms of speed, it only supports 1.25 GT/s [117], which is half of the lane speed of the other links.

The last device depicted in Figure 3.1 is an Intel 82576 dual-port Gigabit Ethernet NIC with support for SR-IOV. Each Ethernet port can be configured to spawn as many as seven VFs. For this evaluation setup, the NIC is configured to provide one VF for the first Ethernet port (eth0: VF0.0) and three VFs for the second port (eth1: VF1.0–VF1.2). The NIC supports four lanes (x4) and a speed of 2.5 GT/s. Depending on the

experiment in Section 3.5, the NIC is either connected to the x4 port of the PCH, or to the integrated PCIe port of the CPU. These two options are mutually exclusive, but the NIC will be connected to the PCH port for most of the experiments. If it is connected to the CPU port, it will be explicitly stated.

The platform's software configuration is based on Ubuntu Linux. Experiments 1, 3 and 4 in Section 3.5 were conducted on Ubuntu Linux 12.10, while experiment 2 was using Ubuntu version 14.04. Each time, QEMU [44] and KVM [43] were employed for virtualizing the system. All guest VMs used the same Ubuntu and kernel version as the host, and each VM was assigned 4096 MB of RAM.

3.4 Evaluation Methodology

In order to quantify DoS attack impact on concurrently running VMs and the host system, three metrics were selected for measurement.

3.4.1 Latencies of 32 bit PCIe Read Transactions

Latencies for single PCIe read transactions give a good general idea about the impact of a DoS attack on the interconnect. Like mentioned in Section 2.3.1.4, a read transaction completely blocks a CPU core until the requested data word arrives from the I/O device. Measuring the execution time of a single read instruction, therefore, results in an accurate report of the round-trip time between CPU and I/O device.

The Time-Stamp Counter (TSC) mechanism provided by the i7 CPU was a suitable choice for the job. It is a 64 bit counter that is initialized to zero at processor reset and afterwards increases monotonically at a known rate, independent of any power saving states that a CPU core might switch between[1]. There is one TSC per CPU core available, and it can be read with a dedicated instruction. The whole routine for measuring the latency of PCIe reads was implemented following the guidelines described in [118].

As a result, measuring was done inside a kernel module that binds to the PCIe device under test. The Linux `readl()` function was used to issue 32 bit reads from PCIe MMIO targets. The module disabled interrupts for the core it was running on, as well as software-based preemption, so that that the measurement routine was not interrupted. Additionally, hardware-based preemption was prevented by disabling HyperThreading in the BIOS. The guidelines in [118] also ensured that the CPU's out-of-order execution capabilities did not falsify the results. Program execution before and after the `readl()` function was serialized to ensure the accurate measurement. Furthermore, dynamic frequency scaling (SpeedStep) and Turboboost were disabled in order to prevent frequency changes of CPU cores depending on the current workload and thermal conditions on the CPU die. This ensured consistent and reproducible results, which would not have been achieved if the CPU cores were constantly switching frequencies between different

[1] This is not guaranteed for older processor generations, and must be checked either by looking into the respective datasheet or by querying the `cpuid` instruction.

measurement runs. All latency results presented in this chapter are the average of one million consecutive samples.

Evaluation of the latency measurement module showed that measurements from within the host OS yielded reasonable and expected results. However, it did not work as intended when run within a VM. One of the problems was that hypervisors virtualize the TSC itself due to various reasons [119, 120]. To give an example: Virtualized operating systems within VMs may use the TSC for internal time-keeping tasks. If these VMs are fully virtualized, they assume to run on the physical CPU for 100% of the time, while in reality, they might be preempted by the host without knowing. Virtualizing the TSC, e.g. by maintaining virtual per-VM copies, can therefore help to prevent errors that may result from this assumption. Additionally, the previously stated feature of the measurement module of turning off interrupts und software preemption to guarantee precise measurements only applies to the virtualized kernel of the VM. If scheduling from the host or arrival of a real interrupt preempts the whole VM, the measurement routine within it is halted as well, and measurements lose accuracy. For the evaluation platform, the effects of TSC virtualization can be demonstrated by measuring the latency of VF0.0 (see Figure 3.1, λ_0) from the host and from a VM with passthrough access to VF0.0. The results are depicted in Figure 3.2.

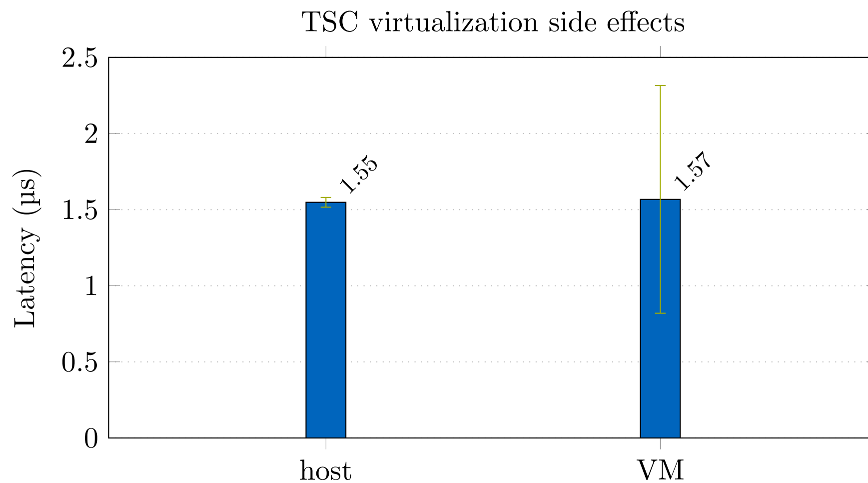


Figure 3.2: The side effects of Time Stamp Counter (TSC) virtualization: Measuring read latencies of VF0.0 from the host yields better results than from inside a VM.

On average, the inaccuracies that are introduced by TSC virtualization cancel each other out. Measurement results from the host and from within the VM show a latency of about 1.5 µs. However, the latter shows a standard deviation drift of 0.75 µs in both directions. In order to circumvent this phenomenon, all latencies within this chapter were measured from the host. The measurement process was always pinned to a dedicated core, so that interferences with VMs were avoided, and a precise measurement of the state of the interconnect was guaranteed.

3.4.2 Network Performance – TCP and UDP Throughput

Network performance in the form of TCP und UDP throughputs is the second metric that was used to measure DoS attack impact in the following experiments. Each of the three NICs of the evaluation platform was incorporated at least once for these tests. Although there is a dependency on the previous metric (latencies for single PCIe packets), the achievable throughputs of the NICs give a better idea of the impact of DoS attacks on real-world I/O workloads.

Throughputs were measured with the `netperf` benchmark [121], which was always executed from within a VM that had passthrough access to either VF0.0 or one of the two other NICs (see Figure 3.1, λ_0, λ_1, λ_2). The respective NIC under test was connected to a dedicated remote machine which acted as sink for the streams. Standard parameters were utilized for both streams, TCP and UDP, which used the default message sizes of 16 KiB and 64 KiB, respectively. Each benchmark was conducted for a period of ten seconds, and all presented results are the average of multiple iterations. No jumbo frames were used, and the Maximum Transmission Unit (MTU) was set to its default size of 1500 bytes.

3.4.3 Storage Performance – SSD Throughput

The third and last metric of the experiments is the storage performance of the host-assigned SSD, which was connected to the evaluation platform's PCH. Therefore, the standard Unix tool `dd` was used to measure the throughput while writing large files to the SSD. These storage performance tests complete the evaluation by adding a second class of real-world I/O that differs from network throughput tests. This helps to further put the impact of PCIe DoS attacks into context.

The results will also emphasize the problems that might arise if future chipsets, which traditionally integrate a platform's storage controllers, start to integrate SR-IOV capable storage controllers. Some server-grade chipsets already do that[2], and next generation storage technologies like NVMe, which is PCIe based, are specifically designed to leverage the opportunities of SR-IOV [84, 85].

3.5 Experiments and Results

This section presents four different experiments that were conducted in order to learn multiple things about DoS attacks on PCIe passthrough: Experiments one and two quantify DoS attack impact by means of PCIe latencies and real-world benchmarks. This is followed by experiments three and four, which investigate parameters that influence DoS attack impact.

[2] Intel C600 series. However, at the time of writing, SR-IOV features of the controller were not yet usable due to missing support in the drivers.

3.5.1 Experiment 1: Attacking SR-IOV Virtual Functions

This experiment investigated what happens if malicious VMs attack their VFs with a PCIe DoS attack. Given the widespread use of SR-IOV NICs in cloud computing, this is the most likely scenario according to the previously introduced threat model, and was therefore selected to be the first experiment. It mainly focuses on the evaluation platform's 82576 dual-port SR-IOV NIC, but also takes the other NICs of the platform into account. Initially, important baseline characteristics of the 82576 NIC are introduced, which was connected to the PCH throughout the experiment (see Figure 3.1).

3.5.1.1 Baseline Performance

The NIC's PCIe connection supports four lanes at 2.5 GT/s with an 8b/10b encoding, which results in a physical layer net bit rate of 8 Gbit/s. This is sufficient to operate both Ethernet ports at full speed simultaneously. Tests with `netperf` showed that each port can achieve 941 Mbit/s for TCP streams and 961 Mbit/s for UDP streams, which is the expected upper limit considering protocol and packet overheads of Ethernet and IP/TCP/UDP. These results were also achieved when running concurrent netperf instances that utilized both ports.

If a port was shared, e.g. by two uncompromised VMs that were attached to VF1.0 and VF1.1 of eth1, respectively, the throughput of netperf halved (if both VMs ran the same test with the same parameters). A third VM would divide it by three, and so on. Considering the intentions behind SR-IOV technology, these are expected results. Furthermore, performance isolation was intact during all netperf tests with VMs that utilized the vendor-supplied driver for their passthrough VFs. Anything that happened on eth0 VFs did not influence the performance of eth1 VFs, and vice versa.

3.5.1.2 Experimental Setup and Results

However, performance isolation between both Ethernet ports can be broken if a malicious VM implements and executes a PCIe DoS attack, like described in Section 3.2, on its own VF. To demonstrate the impact, the following measurements were conducted: VF0.0 of Ethernet port eth0 was attached to an uncompromised VM, and netperf TCP throughputs as well as latencies for reading 32 bit PCIe packets from the VF were recorded (see Figure 3.1, λ_0). This was done while the rest of the system was idle, and while one, two or three PCIe DoS attacks on VFs of the other Ethernet port, eth1, were executed simultaneously. The DoS attacks were launched from distinct VMs, running on dedicated cores and targeted at VF1.0, VF1.1 and VF1.2 (see Figure 3.1, δ_0, δ_1 and δ_2), respectively. The results are depicted in Figure 3.3.

The first column of the barchart, labeled "none", depicts VF0.0's performance during an idle system. Netperf achieved a TCP throughput of 941 Mbit/s, and reading 32 bit words from VF0.0 took 1.58 µs. The next column depicts the results during a single DoS attack on VF1.0 ($A = \{\delta_0\}$). This attack of a single malicious VM sufficed to force a TCP throughput drop from 941 to 684 Mbit/s (-27%). If a second malicious VM was concurrently executing a DoS attack with the first one ($A = \{\delta_0, \delta_1\}$), TCP

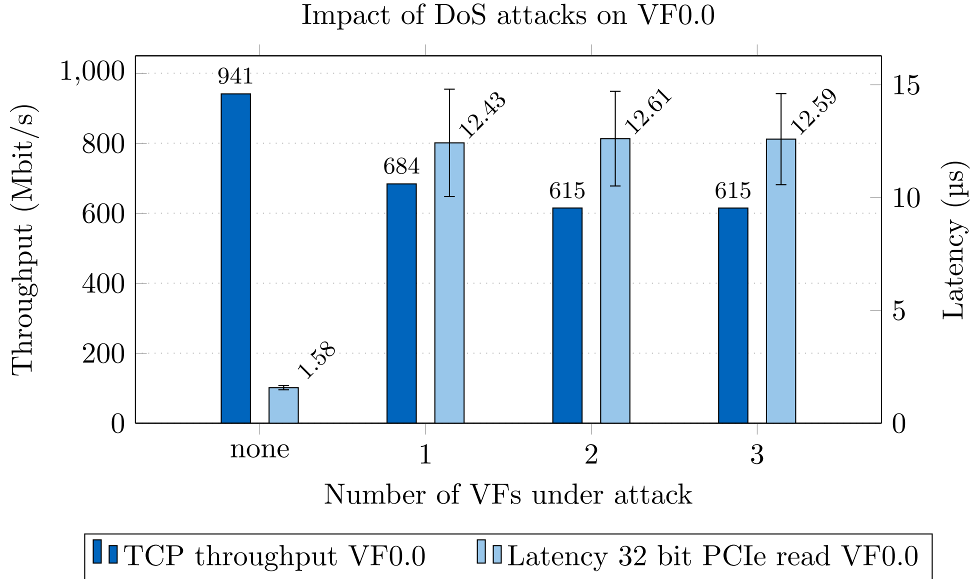


Figure 3.3: Latency for 32 bit PCIe reads and TCP throughput for VF0.0 of the 82576 NIC in an idle system and while one, two or three VFs of the same NIC experience a PCIe DoS attack (see λ_0, δ_0, δ_1 and δ_2 in Figure 3.1).

throughput of VF0.0 dropped further to 615 Mbit/s (-35%). A third DoS attack ($A = \{\delta_0, \delta_1, \delta_2\}$) did not result in any further performance degradation. For that reason, all following evaluations and benchmarks in this chapter will consider two instead of three DoS attacker VMs.

A good hint about reasons for the performance degradation is given by the latency results. During a single DoS attack, it takes about 12.43 µs for reading a single register from VF0.0, compared to 1.58 µs if the system is idle. This is an increase of 687%. A concurrent second attack increases latencies further to 12.61 µs (+698%). With interconnect latencies this high, it is not possible to maintain Ethernet throughputs at one Gbit/s.

These latencies are the result of a congestion on the interconnect that is caused by the DoS attacks. Malicious VMs use their CPU cores to generate PCIe packets at rates much higher than the 82576 NIC can consume or process them. Eventually, this causes the 82576 NIC's ingress buffer to fully saturate. The full buffers, in turn, trigger PCIe flow control mechanisms to halt the upstream device and prevent it from sending more PCIe packets downstream towards the 82576 NIC. The congestion therefore spreads through the whole Q77 PCH until the back pressure reaches the multi-core CPU. Here, it will finally cause two components to block:

1. Every CPU core that executes software that sends PCIe packets to the 82576 (aka the DoS victim). The cores will block on the machine instruction that generates the respective PCIe packet. Blocking stops once a buffer slot in the 82576 was freed and the free slot has propagated upstream.

2. The integrated DRAM controller of the multi-core CPU is affected by the same back pressure mechanisms. It blocks when it tries to transmit completions for DMA reads from the 82576 itself.

3.5.1.3 Second-Order Effects

Considering the conclusions regarding congestion and back pressure, it is evident that there are more PCIe packets affected from blocking than only those that are addressed to the DoS victim device. As PCIe uses a tree topology, it follows that any device that shares PCIe lanes with the DoS victim must itself be affected by the DoS-caused congestion. A good way to prove this assertion was to additionally measure TCP throughputs of the other NICs in the evaluation platform (see λ_1 and λ_2 in Figure 3.1), while the 82576 experienced the same DoS attacks as before. Like the 82576, both NICs are connected to the Q77 PCH and therefore share the lanes of the DMI link with it, as well as an unknown number of resources that implement the undisclosed switching fabric inside the PCH. The results, compared with the TCP throughputs of the previously measured VF0.0, are depicted in Figure 3.4.

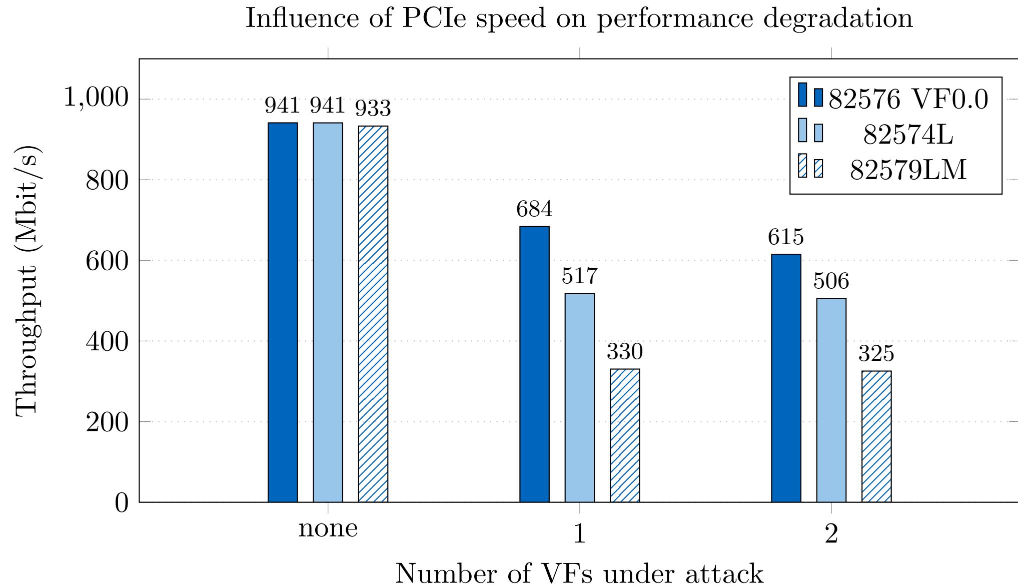


Figure 3.4: TCP throughputs of the three NICs of the evaluation platform, while one or two VFs of the 82576 NIC experience a PCIe DoS attack (see λ_0, λ_1, λ_2, δ_0 and δ_1 in Figure 3.1). The three NICs utilize different PCIe speeds (see Table 3.1).

Compared to VF0.0, the other NICs suffered from even worse performance degradation. For the 82575L, which is the NIC that is hardwired onto the motherboard, throughput dropped from 941 to 517 Mbit/s (-45%) during a single DoS attack. During two simultaneous DoS attacks, throughput dropped to 506 Mbit/s (-46%). The 82579LM, which is integrated into the Q77 PCH, suffered the most of the tree. Its

performance dropped to 330 Mbit/s (one DoS attack) and 325 Mbit/s (two attacks), respectively. This is a decrease of 65% in both cases.

The observed differences in degradation were caused by the different PCIe link speeds that each of the three NICs support. As mentioned in Section 2.3.1.2, PCIe link net bit rates R_{net} are the product of the number of lanes, the physical layer gross bit rate R_b and the encoding. For the evaluated NICs, these parameters are listed in Table 3.1.

Table 3.1: Transfer times for a 1500 byte Ethernet payload.

NIC	Lanes	R_b (GT/s)	Encoding	R_{net} (Gbit/s)	T_{Eth} (µs)
82576	x4	2.5	8b/10b	8	1.5
82574L	x1	2.5	8b/10b	2	6
82579LM	x1	1.5	8b/10b	1.2	10

The table additionally lists the time it takes to transmit a 1500 byte Ethernet payload over the respective link depending on R_{net}. This time is denoted T_{Eth} and is introduced to simplify comparisons between the NICs. The table shows that the 82576 SR-IOV NIC is faster than the 82574L by a factor of four, and faster than the 82579LM by a factor of $6.\overline{6}$. This difference can be crucial if the interconnect is already congested, and explains the differences in degradation in Figure 3.4.

3.5.1.4 Summary

In conclusion, results from this experiment showed that the overall performance degradation of a PCIe device during a DoS attack is influenced by the following two factors:

- The additional latency on the interconnect due to buffer congestion.
- The source-to-sink transmission time (or flight time) of data packets on the PCIe link(s).

If switching intermediaries are incorporated between source and sink device, there might be different link speeds between the individual nodes, which must be taken into account. For instance, (1) CPU to PCH and (2) PCH to PCIe Endpoint. The size of a data packet depends on the actual I/O device type (e.g. the size of an Ethernet frame for Ethernet NICs).

3.5.2 Experiment 2: Influence of the DoS Victim's Processing Speed

The first experiment showed that buffer congestion is one of two major factors that influence performance degradation during DoS attacks. The second experiment was conducted in order to investigate buffer congestion in more detail. Therefore, the influence of the attacked device's processing speed on performance degradation was measured.

3.5.2.1 Fine-Grained Runtime-Configurable Processing Speeds via FPGA

Normally, COTS PCIe devices employ application-specific integrated circuit (ASIC) controllers, which means that processing times for incoming PCIe packets are constant and immutable. Addressing different resources of the same PCIe device might result in different processing speeds, though, but only in a coarse-grained manner. This aspect will be covered shortly in Section 3.5.2.3. However, the following experiment yielded best results and insights with a DoS victim device that had fine-grained control over the processing speed of incoming PCIe packets.

Therefore, a Xilinx SP605 evaluation kit was employed. Its Spartan-6 FPGA was configured to instantiate the vendor-supplied "Integrated Endpoint Block for PCI Express", which enables the FPGA to use the board's PCIe interface. The block is PCIe 1.1 compatible, uses one lane (x1) and supports a link speed of 2.5 GT/s. The block also provides a VHDL example design that supports basic MMIO transactions. This example design was adapted such that it was possible to keep the `wr_busy` signal of the respective interface [122] asserted for a user-configurable time. Thus, it was possible to configure the time the SP605 needed to process a single incoming PCIe packet in a fine-grained manner and during runtime. The minimum achievable processing time was 112 ns.

3.5.2.2 Experimental Setup and Results

The influence of different PCIe packet processing speeds on DoS performance degradation was then investigated with the following setup and measurement series:

1. The SP605 was attached to a VM via PCIe passthrough.
2. For packet processing times between 112 ns and 7.15 µs:
   a) A DoS attack on the SP605 was executed (see δ_3, Figure 3.1).
   b) During the DoS attack, latencies for reading from the 82576 NIC were measured (see λ_0, Figure 3.1).

Latency measurements on the 82576 NIC ensured comparability to the previous experiment. Additionally, it further proved the previous experiment's conclusion that a buffer congestion on a switching device like the PCH affects all its downstream devices. Results are depicted in Figure 3.5.

The results clearly show that the latencies for reading single PCIe packets from the 82576 NIC are directly proportional to the processing time of the DoS target device. Latencies increase for the same reasons as in the previous experiment (see Section 3.5.1.3): The CPU generated PCIe packets, which request a data word from the 82576 NIC, must traverse many of the same buffers and switching resources of the Q77 PCH that are also occupied by DoS-packets that flow downstream to the SP605. Consequently, in a congested condition, any downstream PCIe packet can advance to the next buffer slot only after the DoS victim has processed a packet. It is therefore possible to derive the following heuristic:

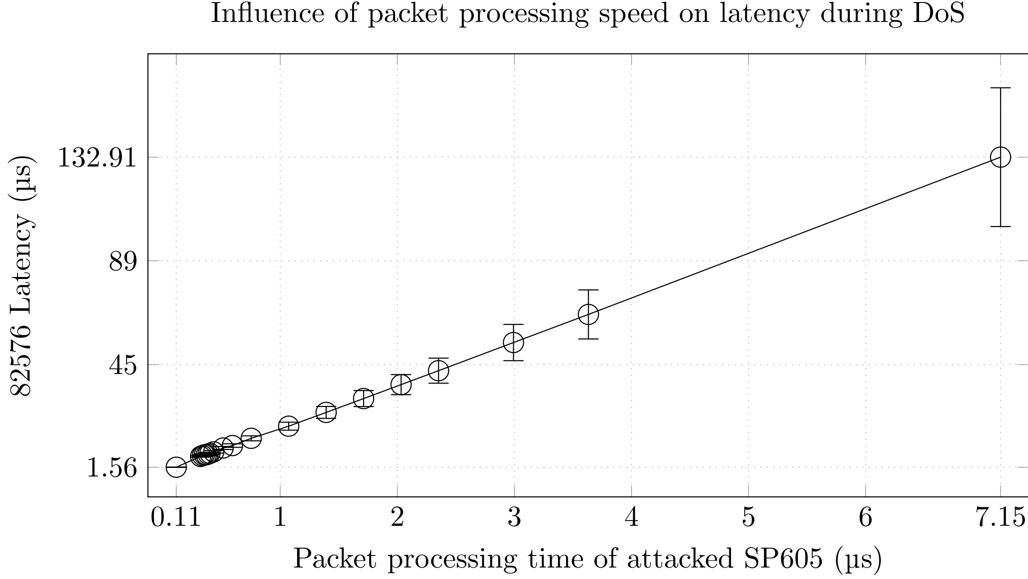


Figure 3.5: Latency degradation for reading 32 bit PCIe words from the 82576 NIC while the SP605 experiences a PCIe DoS attack.

During maximum congestion, the additional latency during a DoS attack Δl_{DoS} for downstream PCIe packets can be roughly estimated as the product of the number of congested buffer slots shared with DoS packets ($N_{\text{buf,DoS}}$) and the DoS victim's processing speed of PCIe packets ($t_{\text{proc,victim}}$). This can be formulated as

$$\Delta l_{\text{DoS}} = N_{\text{buf,DoS}} \cdot t_{\text{proc,victim}}. \tag{3.1}$$

In this experiment, $N_{\text{buf,DoS}}$ was constant because the PCIe topology did not change and only the packet processing times of the SP605 ($t_{\text{proc,SP605}}$) were varied. If this condition is met, it follows that

$$\Delta l_{\text{DoS}} \propto t_{\text{proc,SP605}}. \tag{3.2}$$

Equation 3.1 can therefore be used to model the direct variation between latency results for the 82576 NIC on $t_{\text{proc,SP605}}$, which is depicted in Figure 3.5. Using the same setup, netperf TCP and UDP benchmarks were recorded to complement the latency results and to get a better idea of how these latency numbers translate to real world workloads like network throughput. The results are depicted in Figure 3.6.

The plot shows TCP and UDP throughputs of the 82576 NIC during the same DoS attacks on the SP605. Again, the throughputs are plotted over the packet processing time of the attacked SP605. The results also give a more detailed picture of the dependency between Δl_{DoS} and its impact on networking throughput. The throughput on the y-axis is the result of TCP/UDP payloads that were sent over a time t. The time t includes Δl_{DoS} as well as other components a, which can be formulated as

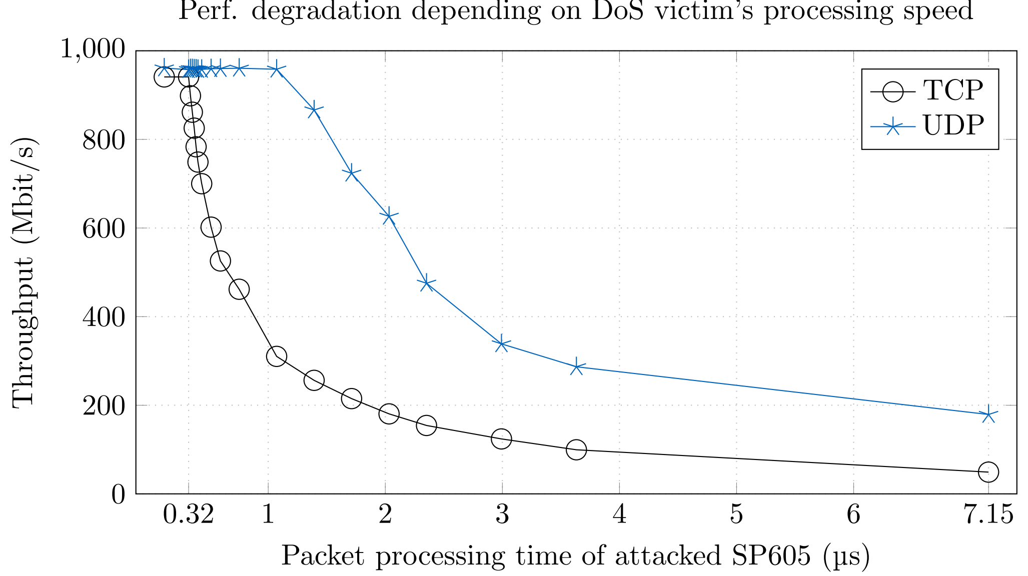


Figure 3.6: TCP and UDP throughput degradation of the 82576 NIC while the SP605 experiences a PCIe DoS attack.

$$t = a + \Delta l_{\text{DoS}} \tag{3.3}$$

If the other components a of t are constant, then it follows that t inherits the linear nature of Δl_{DoS}, which in turn is directly proportional to $t_{\text{proc,SP605}}$ (Equation 3.2). Hence, these dependencies can be formulated as

$$\text{throughput} = \frac{\text{payload}}{t}, \qquad t \propto \Delta l_{\text{DoS}} \propto t_{\text{proc,SP605}}. \tag{3.4}$$

Using this model, throughputs are inversely proportional to the victim's packet processing time, and therefore expected to resemble a rectangular hyperbola (a graph of the form $\frac{1}{x}$) for the parts where Δl_{DoS} is big enough to cause a performance degradation. This is what the results in Figure 3.6 show.

Additionally, Figures 3.5 and 3.6 can be used to derive the maximum latencies that the 82576 NIC can tolerate before TCP or UDP throughputs start to degrade. Degradation for TCP throughputs started for SP605 processing times greater than 320 ns, which translates to a latency of 6.13 µs for the 82576 NIC. UDP throughput, in contrast, degraded if the SP605 needed more than 1.07 µs to process a single PCIe request (82576 latency of 18.9 µs). TCP degradation starts earlier and is worse due to its bigger protocol overhead in contrast to UDP. The latter is connectionless and carries payload data in every packet, whereas TCP has additional SYN and ACK packets that are affected by the congestion on the interconnect as well, but do not count towards the throughput.

3.5.2.3 Processing Speed Variations in COTS Devices

The experiments with the SP605 FPGA device so far showed that slower processing speeds worsen the performance degradation during DoS attacks. Hence, PCIe packet processing speed is a parameter that an attacker wants to maximize in order to achieve his goal of maximum degradation. However, unlike the SP605 FPGA that was used in the previous experiments, it is not possible to dynamically adjust the processing speeds of real-world DoS targets like COTS PCIe NICs. Still, there is a good chance that DoS attacks on COTS devices can be "tuned". Exploration of the evaluation platform's COTS NICs showed that they do not have a uniform processing time for inbound packets, which can be exploited by a DoS attacker accordingly. Like described in Section 2.3.1.3, there is a whole range of addresses per PCIe function in the system memory map. The exact address within the PCIe function's address range finally decides what actual resource inside the PCIe device will be used to store the packet's payload. This is depicted as a high level concept in Figure 3.7.

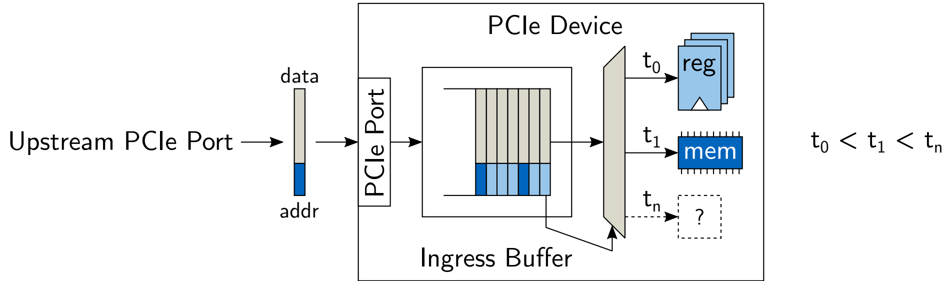


Figure 3.7: Depending on the address of a PCIe packet, it may be processed by or stored into a different resource, which takes different amounts of time.

The block diagram further illustrates that storing a packet's payload may take different amounts of time depending on the resource that stores it. For instance, storing data into registers (t_0) needs less time than storing it into some slower kind of memory (t_1), e.g. DRAM. Most of the time, it is not disclosed what resources lie behind which addresses, so there might also be unknown kinds of storage or packet processing that take even more time (t_n). Consequently, an attacker will want to aim for the address that needs the most time, aka $\max(\{t_0, t_1, \ldots, t_n\})$. In most cases, however, datasheets of COTS devices give no information regarding the processing time depending on the address. Mostly, they only describe the functional aspects of writing to certain addresses inside a device's memory range, so that it is possible for engineers to write device drivers for an OS. Fortunately for the attacker, processing times can be easily estimated in a running system. This process is described in detail in Appendix A.

On top of varying target addresses, processing speeds may additionally differ depending on the payload size (64 bit payloads may be processed slower than 32 bit ones, etc.). A special case of this approach is flooding the device with payload sizes that are not supported by the DoS victim. Depending on the device's implementation of error handling, it may take longer to abort or handle erroneous packets than handling correct ones.

For instance, investigations of the evaluation platform's 82576 NIC, using the method described in Appendix A, yielded the following results. Writes with 64 bit payloads take longer to process than writes with 32 bit. Additionally, there are variances depending on the address of the targeted function. Using 64 bit writes targeting the base address of a 82576 VF's address range resulted in an estimated processing time of 418 ns per PCIe packet. Adding an offset of 0x2800 to the base address, which then targeted the VF's Receive Descriptor Base Address Low (RDBAL) register [108], resulted in about 535 ns. Hence, exploring the device resulted in the discovery of a target address that has an almost 30% slower processing time, which would increase performance degradation accordingly[3].

3.5.2.4 Summary

In conclusion, this experiment first provided a detailed evaluation of the parameters that, during a DoS attack, influence the additional latency Δl_{DoS} of PCIe packets on the interconnect. Results showed that Δl_{DoS} is the product of

- the number of congested buffer slots that must be shared with PCIe packets that target the DoS victim ($N_{\text{buf,DoS}}$) and
- the DoS victim's processing speed of PCIe packets ($t_{\text{proc,victim}}$).

For instance, if a DoS victim is connected to a PCIe switch (e.g. a chipset), then any other PCIe devices also connected to the switch have to share $N_{\text{buf,DoS}} = N_{\text{buf,switch}}$ buffer slots with the DoS victim. If the DoS victim is a multi-function or SR-IOV Endpoint, and the attacker intends to degrade the performance of the Endpoint's other (virtual) functions, then the shared buffer slots are calculated as $N_{\text{buf,DoS}} = N_{\text{buf,switch}} + N_{\text{buf,Endpoint}}$. There might also be egress buffer slots in the CPU's PCIe ports that must be accounted.

If $N_{\text{buf,DoS}}$ is constant, which is true for running systems that only employ COTS hardware, then $t_{\text{proc,victim}}$ is:

- Directly proportional to the latency of PCIe read/write transactions.
- Inversely proportional to the throughput of I/O transactions (e.g. throughputs for TCP/UDP streams).

When attacking COTS PCIe devices, an adversary that wants to "tune" his attack can find the maximum processing time of his victim device by:

- Exploring the whole address range of the victim's PCIe function and finding the address that results in the slowest processing time.
- Checking which needs the most time to process: Packets with big, small or unsupported payloads.

[3] All four experiments of this chapter were conducted (and originally published) before these tuning parameters for DoS attacks on the 82576 were discovered. In hindsight, this resulted in lower-than-possible degradations, but they were still sufficient to prove all the points made.

3.5.3 Experiment 3: Influence of Switching Intermediaries

The previous experiment determined that there are two parameters that have significant influence on performance degradation during DoS attacks: (1) The DoS victim's processing speed and (2) the number of congested buffer slots that must be shared with the DoS victim. The influence of varying the first parameter while keeping the number of buffer slots constant was already quantified in detail in the previous experiment. The following experiment investigated and quantified the effects of having a switching intermediary, like a system chipset, between CPU and DoS victim. The results therefore complement the previous experiment, because this time, the number of congested buffers was varied while the victim processing speed was held constant.

3.5.3.1 Experimental Setup and Results

Both of the previous experiments had DoS victim devices connected to the evaluation platform's chipset, aka the Q77 PCH. For this experiment, the 82576 NIC was inserted into the CPU-integrated PCIe 3.0 slot (see Figure 3.1). The backwards compatibility of PCIe 3.0 allowed the NIC to run with the same speed (x4, 2.5 GT/s) as in the Q77 slot. Thus, the only aspect that changed in contrast to previous configurations was that PCIe packets flowing to the 82576 did not traverse the switching circuitry and buffers of the Q77 PCH. As a consequence, less buffers were involved and therefore less performance degradation was expected during DoS attacks.

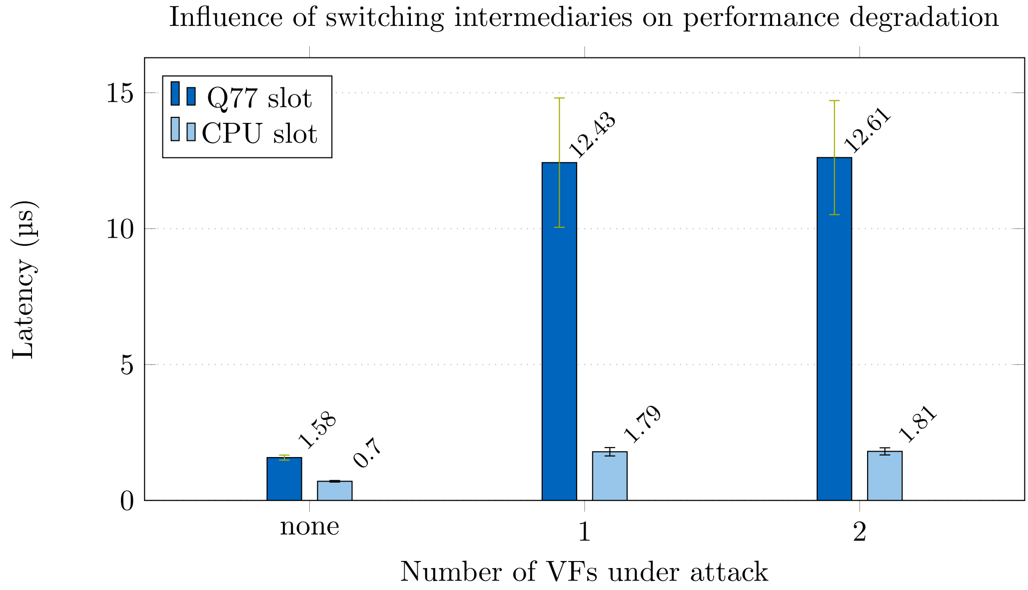
With this configuration, the same measurements and benchmarks as in experiment 1 (Section 3.5.1) were conducted: Latencies and netperf TCP throughputs were measured for VF0.0 during an idle system and while VF1.0 and VF1.1 experienced DoS attacks of malicious VMs. The results of both configurations (Q77 slot in experiment 1 vs. CPU slot in this experiment) are compared and depicted in Figure 3.8.

The results show that removing the chipset from the CPU to I/O device path had great impact on latencies, even in an idle system. Without the chipset as an intermediary, the time needed for reading a 32 bit word from the NIC was less than half the time a configuration with chipset needed. Of course, both latencies were sufficient for achieving peak performance in the TCP streaming benchmark. However, during DoS attacks, there was a significant difference. Latencies for the CPU-slot configuration raised to a maximum of 1.81 µs. In contrast to the 12.61 µs of the chipset configuration, this was still fast enough to achieve peak performance for the TCP throughput stream.

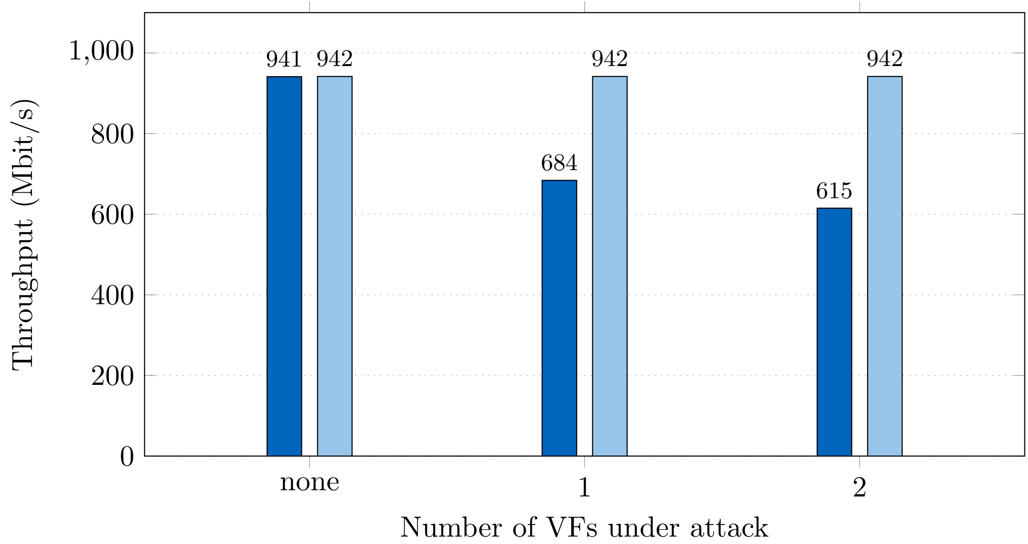
3.5.3.2 Summary

In conclusion, the experiment showed that switching intermediaries may have a significant impact on performance degradation during DoS attacks. A configuration with less buffers is less susceptible to DoS attacks because it results in a lower additional latency Δl_{DoS} when the interconnect is congested (compare Equation 3.1).

It is important to draw the right conclusions from this particular experiment. The CPU-slot configuration did not experience a TCP throughput degradation during the DoS attacks, because the resulting Δl_{DoS} was not big enough to cause one. However,



(a) Latency



(b) TCP throughput

Figure 3.8: Latency for 32 bit PCIe reads and TCP throughput for VF0.0 of the 82576 NIC in an idle system and while one or two VFs of the same NIC experience a PCIe DoS attack (see λ_0, δ_0 and δ_1 in Figure 3.1); Distinguished by PCIe slot (Q77 or CPU).

it cannot be generalized that a CPU-slot configuration automatically prevents performance degradation during DoS attacks by only looking at this particular benchmark. There were numerous parameters involved that, if changed, can also cause a CPU-slot configuration to suffer from performance degradation:

Different hardware Other SR-IOV hardware may have higher maximum processing times for PCIe packets than the 82576 NIC used in this experiment, so that Δl_{DoS} might reach critical dimensions even in a CPU-slot configuration. Additionally, other hardware might provide lower PCIe link speeds, which also impact the amount of degradation.

Different benchmark parameters Netperf was configured to use a large message size (see Section 3.4.2), so that the benchmark was I/O-bound and not CPU-bound. Therefore, blocking of the CPU due to buffer congestion did not steal precious CPU time from the benchmark. Operating netperf with parameters that make it CPU-bound introduces an additional variable that worsens performance degradation. This will be addressed in more detail in Chapter 4.

Different I/O protocols or classes While TCP over Ethernet was unaffected in the CPU-slot configuration of this experiment, there was still a 157% increase in latency. Other transport layer protocols might be more susceptible to additional latency. Furthermore, other classes of I/O than 1 Gbit Ethernet could be employed, which might be less tolerant of additional latencies, e.g. 10 or 40 Gbit Ethernet, Infiniband or disk storage.

3.5.4 Experiment 4: Degradation of Chipset Disk I/O

The three previous experiments demonstrated that TCP/UDP over Ethernet can be susceptible to PCIe DoS attacks. However, network I/O is not the only class of I/O in modern multi-core architectures. In this experiment, the last one of this chapter, both the class of I/O and therefore the benchmark tool that was used were changed. The experiment investigated the effects of DoS attacks on storage I/O, specifically the performance of the host's SSDs, which, among others, store the host OS and all the VM images of the system.

3.5.4.1 Experimental Setup and Results

During the experiment, the 82576 NIC was connected to the Q77 chipset's PCIe slot, where it again functioned as a DoS victim. Thus, the chipset's SATA 3 controller was also affected by interconnect congestion during the DoS attacks (see Figure 3.1). Using the standard Unix tool `dd`, the write throughput that the host can achieve during an idle system and during DoS attacks on the 82576 VFs was measured. In one benchmark run, a total of 1 GiB was copied to the SSD, utilizing four different block sizes that cover most everyday workloads: 512 B, 1 KiB, 512 KiB and 1 MiB. The source data for dd was captured from the kernel's `/dev/zero` pseudo file, so that the SATA controller was only utilized as a sink and not as a source. Results are depicted in Figure 3.9.

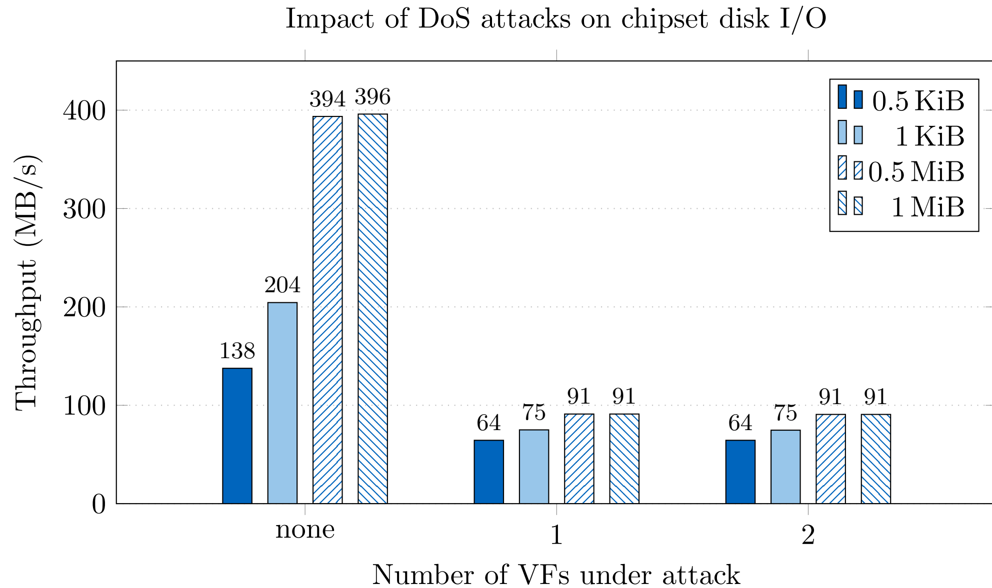


Figure 3.9: Disk throughput for copying 1 GiB of data with `dd` to the platform's SSD, using different block sizes. Results are depicted for an idle system and while one or two VFs of the 82576 NIC experience a PCIe DoS attack (see λ_3, δ_0 and δ_1 in Figure 3.1).

For all block sizes, the results show that a 6 Gbit/s SATA link suffers from severe performance degradation. For large block sizes (512 KiB and 1 MiB), the throughput dropped from 394 and 396 MB/s down to 91 MB/s. This is over four times slower than an idle system, or a decrease of 77%. Utilizing a smaller block size of 1 KiB, throughput degraded from 204 MB/s to 75 MB/s (-63%), which is a slowdown of factor 2.7. The smallest block size, 512 B, degraded from 138 MB/s to 64 MB/s (-55%), which is still over 2 times slower. The results also show that degradation saturates with already one VF under attack, in contrast to the previous network benchmarks where degradation saturated at two VFs under attack.

3.5.4.2 Summary

First of all, this experiment showed that performance degradation caused by PCIe DoS attacks does in fact affect multiple classes of I/O. Besides network I/O (Gigabit Ethernet NICs), modern storage I/O (SATA-based SSDs) might suffer as well. Additionally, the obtained results are also important considering future storage interfaces like the PCIe based NVMe. Like mentioned in Section 3.4.3, future COTS NVMe controllers are likely to adopt SR-IOV, which opens a new attack vector for VMs. If a VM is attached to a NVMe VF, it will be possible to directly attack the storage controller with a DoS attack, and not indirectly like it was done in this experiment.

3.6 System Model for PCIe DoS Attacks on Ethernet NICs

Using the results from the four experiments that were conducted previously in this chapter, an abstract system model for PCIe DoS attacks can be constructed that summarizes the learnings and abstracts from specific hardware and benchmarks. It can therefore give a concise overview about this new class of performance isolation problem and can be used to identify opportunities for mitigating or preventing the DoS problem. For the model, the use case of a generic Ethernet NIC (both legacy and SR-IOV capable) will be used, because at the time of writing, it was the most widespread class of COTS device used for PCIe passthrough. For brevity, the case of direct connection between CPU and NIC is assumed, but switching intermediaries could be easily incorporated subsequently into the model, if needed.

The system model comprises three components: A multi-core CPU, DRAM and an (SR-IOV) NIC. In modern operating systems, communication of sophisticated I/O devices like NICs with software (VMs) is typically handled in an asynchronous manner that includes MMIO access to the I/O device, interrupts and DMA transfers. The system model is depicted in Figure 3.10.

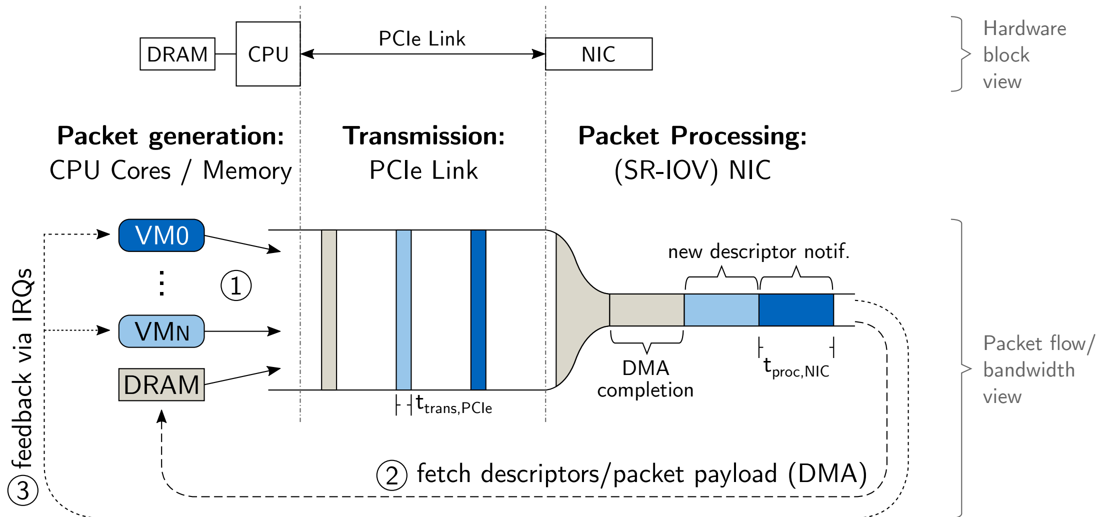


Figure 3.10: System model for legal, closed-loop communication between VMs and (SR-IOV) NIC, using PCIe packets. The bottleneck indicates that packet generation (CPU cores) and transmission (PCIe Link) are both faster than the NIC's packet processing rate. VMs that use a vendor-supplied VF driver incorporate feedback from the NIC and take care of the bottleneck.

The communication in this model is simplified such that it only contains as much detail as needed for presenting the DoS problem. It is based on networking stacks and drivers of modern Linux/Unix operating systems. The model shows multiple VMs running on the virtualized CPU, and each VM is utilizing the vendor-supplied driver for their attached PCIe device or VF. The model in Figure 3.10 only depicts the transmission of Ethernet packets, because DoS attacks primarily degrade the transmission path, not reception.

On initialization, the Ethernet NIC's driver is allocating a bunch of buffer structures in main memory (DRAM). One of these is a ring buffer structure for so-called TX (transmit) descriptors. A TX-descriptor is a data structure that can be understood and processed the NIC. It contains the main memory address of a ready-to-send Ethernet packet as well as its size and maybe other metadata that is optional. Once the TX-ring buffer is allocated, its start address is communicated to the NIC. Afterwards, Ethernet packet transmission is handled as follows:

1. Operating System and driver prepare and compile an Ethernet packet and place it into main memory. Then, a corresponding TX-descriptor is built and added to the driver's TX ring buffer. Finally, the driver notifies the NIC about new entries in the TX ring via PCIe. For this step, NICs usually provide a dedicated memory location (e.g. a special register) to which the driver, which executes on the CPU, can write via MMIO. In Figure 3.10, writing to this register is marked as step (1).

2. After the NIC received and processed the notification about new TX descriptors, it fetches the new descriptors from the system's main memory via DMA. The fetched descriptor is then analyzed and a subsequent DMA transaction fetches the Ethernet packet contents from the main memory address to which the descriptor pointed. In Figure 3.10, both steps are summed up as step (2). Finally, the NIC is able to schedule the packet for transmission over Ethernet.

3. After successful transmission of the packet, the NIC notifies the driver. This is usually done with via interrupts and depicted as step (3) in Figure 3.10. Most likely, before sending the IRQ, the NIC has first updated a data structure in main memory that contains the number of transmitted packets and other relevant statistics. This step is not depicted for brevity.

This closed-loop communication scheme ensures that a NIC driver is operating the NIC hardware within the bounds of its capabilities. The feedback mechanism comprising statistics and IRQs ensures that a CPU does not overstrain the NIC and flood it with more requests than it can serve. Section 3.5 showed that this is a vital aspect, because in typical x86-based systems, CPUs and PCIe links are able to deliver PCIe packets faster than a NIC can process them. This is depicted by the bottleneck at the transition of PCIe link to NIC. Of course, as shown in Section 3.5.2.3, there is not a uniform processing speed for all PCIe packets, so that the bottleneck varies depending on the destination of the packet. This is omitted in the figure in order to provide a concise depiction.

If this bottleneck is exploited by malicious VMs with DoS attacks as described in Section 3.2, which flood the NIC with spurious PCIe write packets, a congestion on the interconnect will emerge. The resulting backpressure then causes performance degradation, which depends on a number of parameters that were presented in Sections 3.5.1.4, 3.5.2.4 and 3.5.3.2. The model with a malicious attacker VM and a congested interconnect is depicted in Figure 3.11.

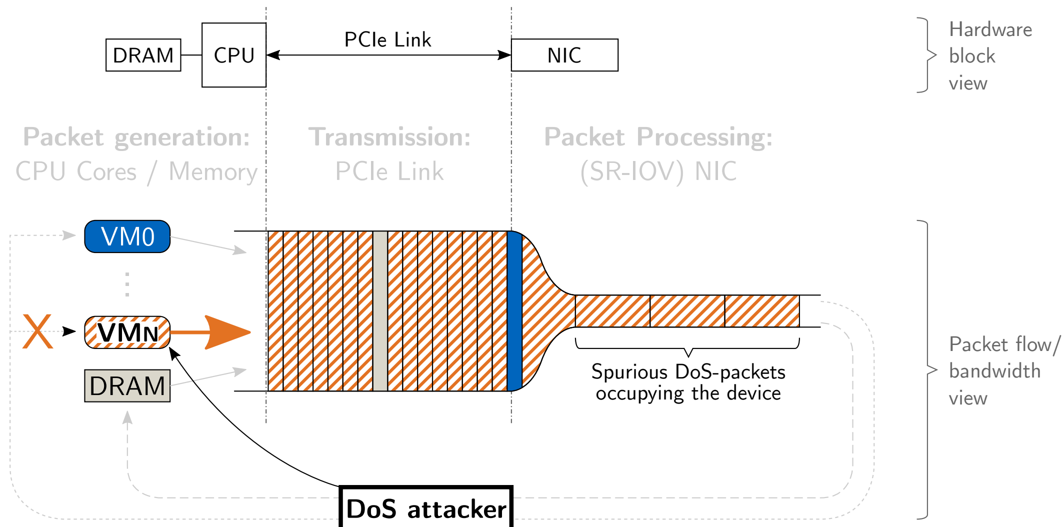


Figure 3.11: A malicious VM exploits the bottleneck between packet generation/transmission and processing. It floods the NIC with spurious PCIe write packets and does not listen to feedback from the device. As the NIC cannot process the packets faster than they arrive, backpressure builds up that also delays transactions from concurrent VMs and DRAM, eventually causing performance degradation in the system.

The figure of the model depicts how downstream PCIe packets get delayed when the interconnect is congested. The delay affects packets from other VMs which run on different CPU cores than the attacker as well as any DMA requests that the DRAM controller completes via PCIe. A third instance that is not depicted for brevity is the hypervisor, which also runs on CPU cores. Its packets also suffer from congestion delays, so that downstream I/O devices that are attached to the host might see performance degradation as well.

3.7 DoS Attack Classification

The results of Section 3.5 showed that DoS attacks can cause performance degradation at different locations of the PCIe tree. For example, degradation can affect PCIe functions of the same physical device, functions of different physical devices, and more. In the evaluations of Section 3.5, degradations were identified by comparing baseline results of specific network or disk I/O benchmarks for specific I/O device classes with results during DoS attacks.

In this section, these observations are generalized and abstracted from specific benchmarks and device classes. A classification is introduced that allows easy characterization of DoS attacks and helps to quickly communicate their impact on the components of a system. Therefore, the following hierarchy of classes and sub-classes of PCIe DoS attacks shall be defined:

1. Inter-device attack

2. Intra-device attacks
   a) Inter-function attack
   b) Intra-PF attack
   c) Inter-PF attack

Some of these attacks are restricted to specific kinds of PCIe devices. It will be differentiated between (1) single-function legacy devices, (2) multi-function legacy devices, (3) single-function SR-IOV devices and (4) multi-function SR-IOV devices. Theoretically, it is also possible to incorporate legacy functions and SR-IOV PF/VF functions onto the same physical PCIe device, but this unlikely combination will not be explicitly mentioned. However, descriptions of the attacks can be applied to such devices as well. In the following, the attack classes are described in detail. Additionally, it is discussed which criteria define if flooding a passthrough function is either a nonsensical, but still legal use of it, or already a DoS attack.

3.7.1 Inter-Device Attack

An inter-device attack describes PCIe DoS attacks that affect other physical I/O devices of the system. It is sufficient to see any amount of performance degradation on another physical device in order to classify an attack as inter-device DoS attack. Here, the most likely case is that an attacked PCIe device is connected to any kind of (switching) intermediary that connects to other devices as well. Like shown in Section 3.5.3, a system's chipset would be a common example of an intermediary in x86 systems. Inter-device attacks can be executed on any kind of PCIe device, be it single- or multi-function, legacy or SR-IOV. Devices that see performance degradation due to such an attack may also be of any kind. An example is depicted in Figure 3.12.

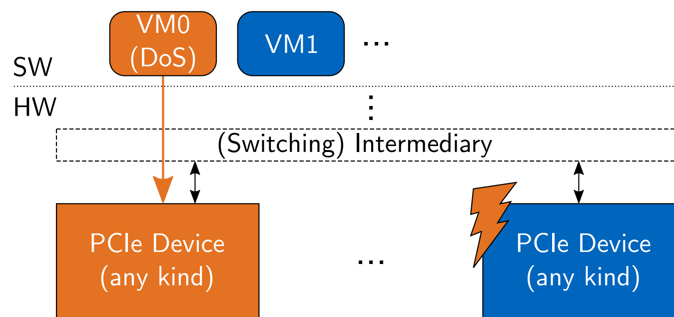


Figure 3.12: Inter-device attack. A DoS attack on a PCIe device causes performance degradation to other PCIe devices of the PCIe tree.

3.7.2 Intra-Device Attacks

Intra-device attacks describe a super-class of attacks that applies to any kind of multi-function PCIe device. In contrast to inter-device attacks, performance degradation emerges on the same physical I/O device that is attacked by a malicious VM. This class contains three sub-classes of attacks, which are described in the following.

3.7.2.1 Inter-Function Attack

Inter-function attacks apply to multi-function legacy PCIe devices. Suppose two VMs, VM0 and VM1, are attached to different functions of the same legacy PCIe device. If VM0 attacks its function, and VM1 experiences any amount of performance degradation while using its own function, then VM0's attack can be classified as an inter-function attack. A block diagram of such a scenario is depicted in Figure 3.13.

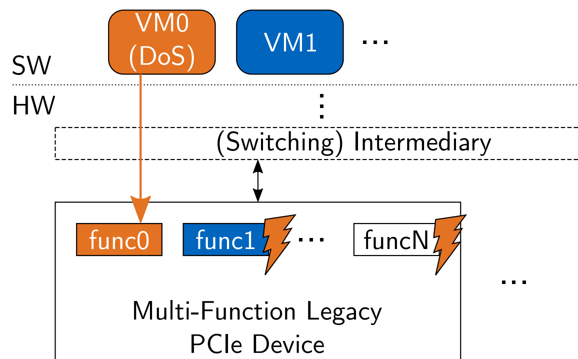


Figure 3.13: Inter-function attack. A DoS attack on one function of a multi-function PCIe device degrades the performance of other functions of the same device.

3.7.2.2 Intra-PF Attack

Intra-PF attacks describe DoS attacks that apply to any kind of SR-IOV device (single- or multi-function) and affect VFs that belong to the same PF. Therefore, suppose a system with an SR-IOV device like depicted in Figure 3.14. PF0 is used to spawn two VFs, VF0.0 and VF0.1, which are attached to two virtual machines, VM0 and VM1, respectively. Both VMs utilize their attached VFs at the same time. For an ideal SR-IOV device, performance of whatever class of I/O is provided by the PF would be split evenly, so that each VM gets a 50% share. In case VM0 becomes a DoS attacker and starts to flood its attached VF0.0, this action is a DoS attack only if it causes VM1's performance to degrade more than 50%. If the flooding would cause VM1's performance to degrade less than 50%, then VM0's action cannot be considered a DoS attack. Instead, it is only a nonsensical, but harmless use of the resources of its own attached VF. In more general terms, flooding an SR-IOV VF is a successful intra-PF attack only if the flooding causes the performance of another VF, which belongs to the same PF, to degrade more than legal sharing of the attacked VF would degrade it.

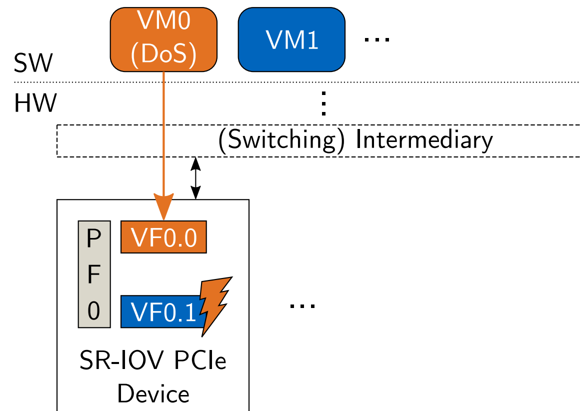


Figure 3.14: Intra-PF attack. A DoS attack on one VF of a PF degrades the performance of other VFs of the same PF more than legal sharing of the attacked VF would do.

3.7.2.3 Inter-PF Attack

Inter-PF attacks describe DoS attacks that affect VFs that belong to different PFs. To give an example, suppose a system like depicted in Figure 3.15. VM0 is again attached to VF0.0, but this time, VM1 is attached to VF1.0. In this setup, simultaneous operation of both VFs would yield 100% of PF performance for each VM, because only a single VF is instantiated per PF. Here, performance isolation requirements demand that VF0.0 of PF0 cannot have any performance impact on VF1.0 of PF1, because the VFs belong to physically disjunct I/O resources. In general terms, flooding an SR-IOV VF is a successful inter-PF attack if the flooding causes any amount of performance degradation for VFs that belong to different PFs. A block diagram for this example is depicted in Figure 3.15.

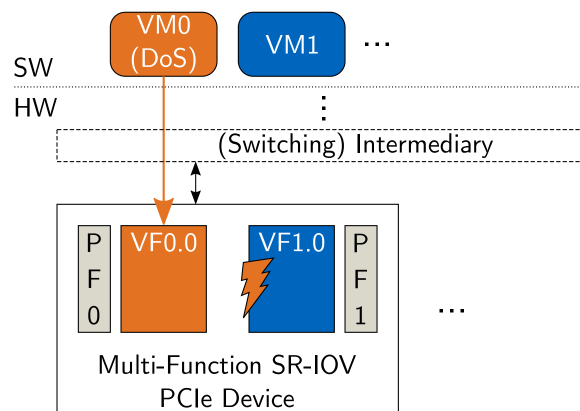


Figure 3.15: Inter-PF attack. A DoS attack on a VF of PF0 degrades the performance of a VF of PF1.

3.7.3 Combinations

The classifications are not mutually exclusive. A single DoS attack can turn out to be a combination of multiple of the presented classes. For instance, consider a virtualized system like depicted in Figure 3.16.

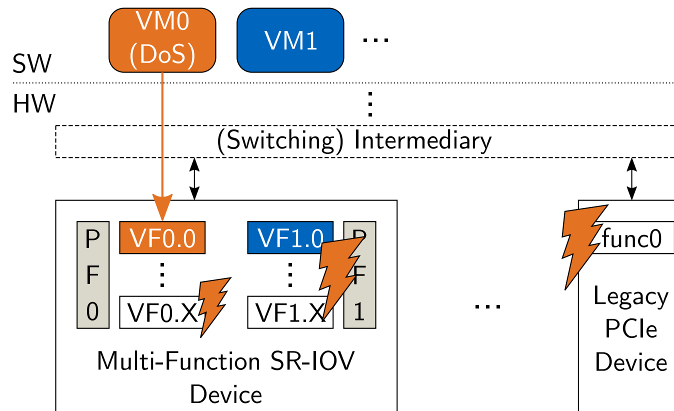


Figure 3.16: A DoS attack on VF0.0 degrades performance of other VFs of the same PF, of VFs of a different PF, and another physical I/O device.

If a DoS attack on VF0.0 of the multi-function SR-IOV device degrades performance of VF0.X for more than 50%, and any VF1.X also sees some degradation, then the attack is an intra- and inter-PF attack. There may also be another physical I/O device in the system that also experiences performance degradation due to the DoS attack on VF0.0. Then the attack must be classified as an intra- and inter-PF as well as an inter-device attack. But it could also be possible that VF0.X degrades for less than 50% but VFs1.X still see some degradation. Then the attack would be an inter-PF and inter-device attack only. In any case, the impact of a DoS attack is a complex combination of a multitude of hardware and software parameters as well as the PCIe topology of the system.

3.8 Summary

PCIe passthrough and SR-IOV are technologies that enable hardware-assisted I/O virtualization for (untrusted) virtual machine guests. They are implemented in commercial off-the-shelf products and corresponding software support is implemented in the respective operating system kernels and hypervisors. The technology convinces with low latency and near-native performance. It has been embraced by cloud computing providers like Amazon and is considered a viable option for I/O virtualization of future embedded and mixed-criticality multi-core systems.

This chapter introduced that PCIe passthrough and SR-IOV suffer from performance isolation issues that can be exploited by DoS attacks on PCIe passthrough MMIO resources. First, a threat model was introduced that covers the cloud computing and embedded virtualization domains. Both are vulnerable because they either provide

passthrough access to untrusted users as part of the business model (cloud: IaaS) or by exposing potentially exploitable interfaces to untrusted users (cloud: PaaS, embedded: Infotainment). Subsequently, the implementation of DoS attacks is discussed and an exemplary implementation in the C language is presented.

In order to investigate and evaluate performance isolation issues of PCIe passthrough and SR-IOV, an evaluation platform was compiled. The platform is an x86 based system comprising a dual-port SR-IOV capable Gigabit Ethernet adapter, two single-port non-SR-IOV Ethernet adapters, an SSD disk and an FPGA board with custom logic design to aid the evaluation. On this platform, four different experiments were conducted which provided results that contributed to a thorough understanding of the isolation issues during PCIe DoS attacks. The utilized benchmarks included network streaming benchmarks for TCP and UDP protocols, latencies for single PCIe reads and storage throughput for SSDs.

Results showed that DoS attacks can significantly degrade the performance of SR-IOV devices as well as other PCIe devices on the interconnect. For instance, experiment 1 demonstrated that, during DoS attacks, Ethernet throughputs for the SR-IOV adapter drop by up to 35% and latencies increase up to 698%; Experiment 4 additionally demonstrated that disk storage throughput drops by up to 77%. The actual amount of degradation always depends on a multitude of hardware- and software parameters of the system, as well as the PCIe topology. The two most influential parameters are (1) the PCIe packet processing speed of the device that is targeted by the DoS attack, because it influences the additional latency caused by buffer congestion, and (2) the number of PCIe packet buffers that an affected device must share with the DoS victim device.

In conclusion, the results of the experiments were used to introduce an abstract system model for PCIe DoS attacks on Ethernet devices. The model abstracted from the specific hardware and benchmarks used in the experiments and therefore provided a generic presentation SR-IOV's performance isolation issues. Finally, the chapter was closed with the presentation of a classification of DoS attacks. The classes that were introduced are distinguished by their reach inside the system and create new vocabulary that helps to characterize PCIe DoS attacks in future research.

4 Mitigating Performance Isolation Issues Through Monitoring and Scheduling

The previous chapter showed that the design of PCIe passthrough and SR-IOV is exploitable by PCIe DoS attacks, which may break a virtualized system's performance isolation. As PCIe passthrough and SR-IOV are based on hardware-assist, adding facilities for complete isolation would require significant changes to existing hardware. Additionally, sophisticated hardware changes might not be transparent to system software. However, the need for complete isolation strongly depends on the domain that utilizes passthrough technology. For the embedded and mixed-criticality domain, complete isolation is mandatory if real-time requirements are in place. Chapter 5 will present suitable, hardware-based solutions for these domains. In cloud computing, however, it is adequate to sufficiently mitigate a PCIe DoS attack until an operator can inspect the offender VM and take appropriate measures. Mitigation approaches offer the advantage that their implementation can be realized in a lightweight and low-overhead manner.

This chapter presents a solution that utilizes lightweight hardware and software extensions to mitigate performance isolation issues for the cloud computing domain. Section 4.1 starts by defining the goals and requirements of the intended solution. Afterwards, Section 4.2 introduces a system design that fulfills the goals and requirements. A prototype implementation of the design on an x86 system is presented in Section 4.3, and is subsequently evaluated with typical cloud computing benchmarks in Section 4.4. Finally, the chapter is summarized in Section 4.5.

4.1 Goals and Requirements

In the following, fundamental goals and requirements are introduced for mitigating the performance isolation issues of PCIe passthrough and SR-IOV for the cloud computing domain. The requirements are also defined with portability in mind, so that they do not restrict the resulting design to cloud computing, but make it portable to similar domains like high performance computing or datacenter computing as well.

4.1.1 Goals

The overall goal is to develop a design that extends existing cloud computing architectures with facilities that enable detection and mitigation of performance isolation breaches in passthrough hardware. In order to integrate well, the design should seek synergies with existing approaches and technologies that counteract other malintent actions (e.g. spoofing network addresses) of VMs. It should not only be compatible to the

prevalent combination of processor architecture and I/O class (x86 and Ethernet), but be generic enough to be easily applicable to other combinations as well.

Most importantly, the design must adhere to the paradigm of PCIe passthrough and SR-IOV, which is to keep the host out of the VM-to-I/O-device data path for the sake of performance. Any design that would rely on host software that interrupts, inspects and forwards MMIO PCIe communication of VMs in order to detect DoS attacks must therefore be ruled out. This would ultimately push a large part of the system back to paravirtualization and break with the passthrough paradigm. Hence, information about communication between VMs and passthrough functions must be collected in a non-intrusive manner, for example with hardware-assist that operates concurrently to normal VM execution.

However, detected and on-going DoS can be mitigated by trusted host software without breaking the passthrough paradigm. This is because it is possible to influence the flooding of a malicious VM by adapting the host's scheduling of the (QEMU) process in which the VM is executing.

4.1.2 Requirements

Non-intrusive detection: DoS detection facilities must not interrupt the direct communication of VMs with their respective passthrough functions. Any information that is needed in order to detect DoS attacks must be collected in a non-intrusive manner.

Low overhead detection: DoS attacks will be rare events. Therefore, detection should impair the performance of VMs or the host as little as possible.

Secure detection: The detection of attackers must be tamper-proof, i.e. detection facilities must be accessible by privileged host software (hypervisor) only.

Unambiguous identification of attackers: Often times, virtualized multi-core systems in cloud computing have a high degree of dynamics in terms of scheduling. VMs may frequently switch cores and/or execute on multiple cores at the same time. The design must cope with these dynamics and unambiguously identify attacker VMs.

Sufficient mitigation: On-going PCIe DoS attacks must be sufficiently mitigated until an operator or software watchdog takes appropriate measures. Here, sufficient mitigation means that the amount of performance degradation during DoS attacks is capped to a reasonable minimum so that appliances running on the respective machines stay close to baseline performance.

Scalability: Detection facilities must be scalable, so that they support the allowed range of PCIe (virtual) functions according to the standard.

4.2 Design and Exploration

This section presents a high-level design for a solution that adheres to the previous section's requirements. The design enables DoS detection through non-intrusive, hardware-assisted monitoring of PCIe transactions, and mitigates on-going DoS attacks with software-based countermeasures from the trusted host (e.g. the hypervisor). It does therefore not break with PCIe passthrough's paradigm of no host involvement in the VM-to-I/O data path. A block diagram of the design is depicted in Figure 4.1.

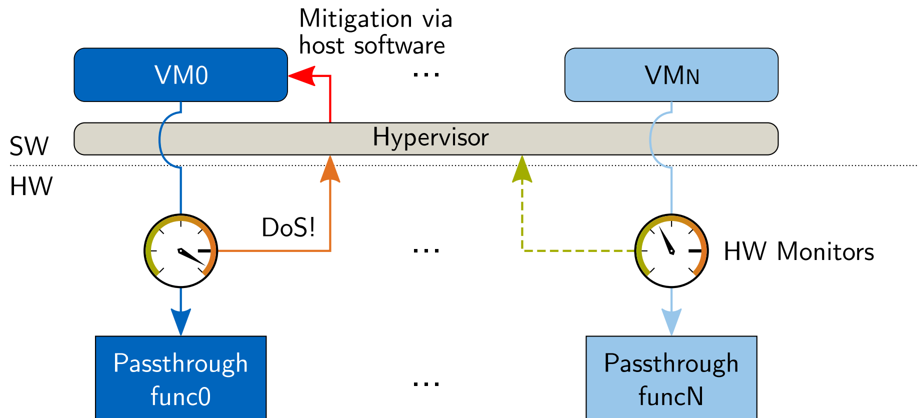


Figure 4.1: High-level design for a solution that utilizes hardware monitoring for DoS detection and host software for mitigating PCIe DoS attacks of malicious VMs. Passthrough functions can be either legacy PCIe functions or SR-IOV VFs.

4.2.1 Monitoring and DoS Attack Detection

The design is based on the observation that DoS attacks always create larger PCIe write transaction rates (wr/s) in the CPU-to-passthrough direction (MMIO) than even the most demanding legal use of a passthrough function. Therefore, it is possible to determine a threshold PCIe transaction rate, which, if surpassed, indicates a DoS attack. The reasons for this are elaborated in the following.

As presented in Chapter 3, DoS attacks work because the attacker is enforcing a constant congestion over time on the interconnect. The congestion emerges because the MMIO resources of the passthrough function are overloaded, aka they receive packets at faster rates than they can process them. This clearly is an unwanted operating point of the passthrough function, which was never intended to be reached in the first place. Hence, any software that is not malicious will always produce PCIe transaction rates that are below the rates of a DoS attack in order to avoid the unforeseeable consequences of a congestion. This difference between legal and malicious transaction rates allows to define a threshold value for PCIe wr/s that distinguishes DoS attacks from legal function use. As this threshold value depends on the actual hardware, it must be determined individually for each system, which will be addressed in detail in Section 4.4.3.

In order to realize this approach towards DoS detection, hardware monitoring facilities are needed that satisfy the following requirements.

1. Monitors must be able to detect and count CPU-to-passthrough PCIe transactions (MMIO) at passthrough function granularity. Here, a passthrough function means either a classic function of a legacy PCIe device or a VF of an SR-IOV capable device.

2. To satisfy the requirement for secure detection, any access to monitoring facilities (setting parameters, reading out values, etc.) must be restricted to privileged host software.

DoS detection can then be implemented by sampling the monitor's counter value for MMIO transactions, calculate the transaction rate, and compare it to the threshold value for DoS attacks. Ideally, monitoring facilities already support programmable sampling and comparison features so that this task can be offloaded to hardware, which reduces CPU overhead. However, this is not a strict requirement as long as the CPU overhead for these routines is acceptable.

4.2.2 Exploration of Monitoring Alternatives

In a PCIe network, it is possible to realize hardware monitoring in three different kinds of components. (1) The CPU cores, which are the sources of the PCIe packets, (2) the I/O device that provides the passthrough function(s) or, if existent, (3) an intermediary device like a chipset or switch that is interposed between CPU and I/O device. In the following, all three possibilities are explored and checked against the requirements that were proposed in the previous sections.

4.2.2.1 Monitoring in CPUs

Nowadays, common CPU architectures and their respective processors already provide so-called Performance Monitoring Units (PMUs), which enable monitoring of numerous architectural events in a CPU. For example, they are available in IBM POWER8 [123], ARM Cortex [124] and most Intel processors [4]. The following exploration will only cover Intel processors, because at the time of writing, cloud computing providers paired PCIe passthrough technology almost exclusively with Intel (Xeon) processors.

In each CPU core, there is one PMU, which is a collection of registers that enable performance monitoring. Each PMU provides a set of Performance Monitoring Counter (PMC) registers (`IA32_PMCx`), which count events that are selected via corresponding selection registers (`IA32_PERFEVTSELx`). The bulk of supported events provides insight to the cache and memory subsystem. Certain complex events also need further specification via associated configuration registers. For instance, if `IA32_PERFEVTSEL0` is configured to select so-called offcore events for counting, `MSR_OFFCORE_RSP0` must further specify the exact offcore event that shall be counted by `IA32_PMC0`. Offcore events include, among others, I/O and MMIO transactions. A short overview of PMUs is given in Figure 4.2.

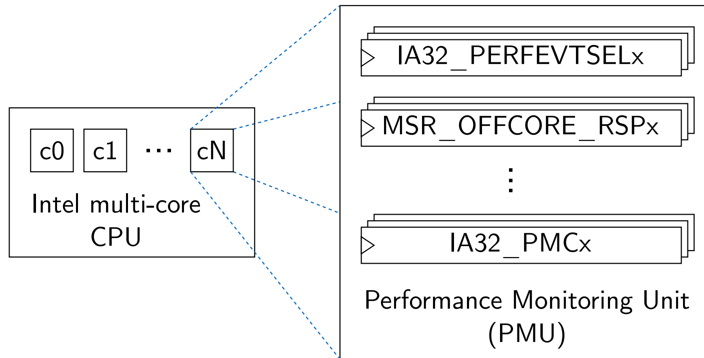


Figure 4.2: Overview of performance monitoring facilities in modern Intel CPUs. Each core provides multiple Performance Monitoring Counter (PMC) registers, which count certain architectural events like cache misses, I/O transactions and many more. The specific event on which a counter triggers is selected by corresponding event selection registers (a full list of supported events is found in [4]).

PMCs can also be configured to generate interrupts when they overflow, enabling "interrupt-based event sampling" [4]. In order to trigger overflows after a certain number of events, counters are preset to a modulus value. On overflow, a performance monitoring interrupt (PMI) is generated, which invokes an interrupt service routine that reacts on the condition and resets and restarts the counter.

Exploration of the Intel PMUs lead to the conclusion that they are not (yet) perfectly suited for the task at hand. First, the event selection lacks granularity. Although an event selector for non-DRAM system addresses exists, which include MMIO transactions, it is not possible to differentiate MMIO transactions by filtering for target addresses. This means that non-PCIe MMIO is counted as well as PCIe transactions to multiple passthrough devices. For example, detecting DoS attacks of VMs with more than one passthrough function, e.g. Ethernet and GPU, is not possible. Therefore, the requirement for distinguishing PCIe packet flows at passthrough function granularity cannot be satisfied. Second, obtaining PCIe transaction counts from PMCs requires either interrupt-based sampling or polling of a PMC for each CPU core in the system. Afterwards, the transaction rates must be calculated and compared to a threshold value. This puts extra load on the CPU, which, depending on the efficiency of the implementation, may violate the requirement for low overhead detection. If a malicious VM executes on multiple cores simultaneously and distributes the attack on its multiple cores, DoS attack detection becomes even more complex. In this case, detection software must merge and evaluate sampling data from all cores, which further increases CPU overhead. Third, PMUs are limited resources that cannot be reserved. Detection software must therefore either ensure that conflicting use of PMUs with other system software can be ruled out or that PMUs are shared properly [125].

On the positive side, the requirement for secure detection is satisfied by current PMU hardware, because access to the registers via VMs can be restricted and emulated. The host can therefore prevent manipulation by malicious VM software. If future PMUs sup-

port fine grained address filtering of MMIO transactions, and the software implementation of sampling and threshold comparison can be realized with acceptable performance overhead, they become a viable choice for DoS detection

4.2.2.2 Monitoring in I/O Devices

Like in CPUs, there is support for performance monitoring in modern, sophisticated I/O devices. For instance, the SR-IOV capable XL710 10 Gbit Ethernet controller provides performance and statistics counter for the PCIe link [126]. Unfortunately, again, like in CPUs, it is not possible to differentiate MMIO transactions at passthrough function granularity. However, the XL710 shows that little changes to existing hardware would enable the needed granularity, as well as DoS detection inside the I/O device. A proposal for a high-level design that satisfies the monitoring requirements is depicted in Figure 4.3.

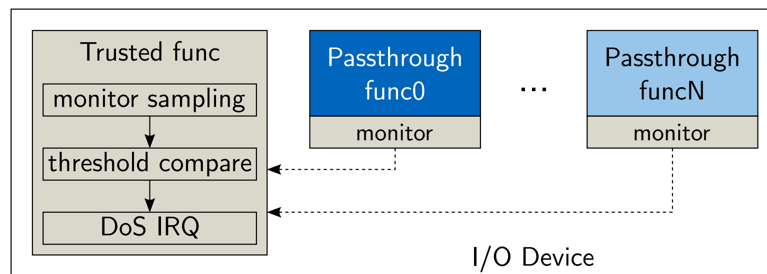


Figure 4.3: A proposed design for monitoring and detecting DoS attacks on passthrough functions inside the I/O device. A trusted, host-managed function samples incoming transactions of the passthrough functions and compares their transaction rates to a threshold value, which is programmed by the host. If the threshold is surpassed, an interrupt is sent to the host, informing it about the DoS attack.

Monitoring MMIO transactions at function granularity inside the I/O device is a straightforward process. Since incoming PCIe packets must be routed to their target function anyways, routing information can be used to implement lightweight per-function monitors that satisfy the granularity requirement. This also simplifies the detection of DoS attacks that are distributed between multiple CPU cores, because monitoring happens at the target function, where the transactions join. Once the DoS victim function is known, the attacker VM can be easily identified with a lookup in the host, because the assignment of passthrough functions to VMs is static.

In order to secure access to monitoring data, a trusted function in the I/O device is needed that is managed by the host. For SR-IOV devices, this function already exists in form of the PF, whereas classic multi-function PCIe devices could provide a dedicated function for the host. Besides access to raw monitoring data, it is possible to implement DoS detection inside the trusted function as well. This can be realized by sampling the monitors and comparing the transaction rates of the passthrough functions to a threshold value that was programmed by the host. If one or more of the functions surpass the threshold, an IRQ is sent to the host, which can then poll the trusted function in order to check which passthrough function is under attack.

Another aspect in favor of DoS detection inside the I/O device is that modern hardware already provides facilities for detecting malicious VM activities. This hardware could be shared where possible. For instance, recent COTS Ethernet server adapters offer a feature called Malicious Driver Detection (MDD) [109]. MDD performs numerous checks on data that is supplied by VMs, e.g. anti-spoofing checks on MAC addresses and VLAN tags, as well as numerous validity checks on transmit descriptors, like checks for correct TCP and UDP header sizes. If malicious actions are detected, the host can be informed by sending an interrupt. DoS detection facilities like described in Figure 4.3 could be implemented alongside of existing MDD hardware and share resources where possible, e.g. interrupt facilities for notifying the host, or registers that log what kind of misuse happened. Also, an integrated detection would not require any CPU cycles, so that the requirement for low overhead detection is satisfied.

4.2.2.3 Monitoring in Intermediaries

The third possibility is to monitor MMIO transactions in intermediaries like the system's chipset (aka the PCH). However, common PCHs for Xeon processors do not provide any sophisticated monitoring hardware. In theory, implementation in intermediaries needs the most complex filtering functionality of the three alternatives. Monitoring hardware must filter MMIO transactions for their destination system address (identifying the passthrough function), and at the same time, determine some kind of source identifier, e.g. the CPU core ID that issued the transaction. The latter is needed to identify the source VM. Additional DoS detection can be realized either by sampling the monitors in software (like proposed in Section 4.2.2.1) or add detection in hardware like proposed in the previous section.

4.2.3 Mitigation

After a DoS attack has been detected by any of the three monitoring and detection solutions that were described in the previous sections, it must be mitigated with software countermeasures. Here, the cloud computing provider is free to choose any approach that works best for it, because tenants have to agree to the terms of use of the provider prior to using its cloud instances. Hence, the kind of countermeasure that is taken in case of a DoS attack can be communicated beforehand. In the following, two pragmatic approaches are introduced and explained in detail.

4.2.3.1 Isolation via Freeze or Migration

A straight forward solution is letting the host freeze the attacker VM. Without executing on a CPU core, no more PCIe DoS packets can be generated that congest the interconnect.

If further investigation is desired, it is possible to migrate the respective VM to an isolated host. Here, the attacker VM can be resumed without harming concurrent VMs, and an operator of the cloud computing provider could start to further investigate the cause of the attack. For example, results of an investigation could be malicious, harmful code

or a buggy device driver. Finally, the result of the investigation can be communicated to the VM tenant. However, it must be remembered that live migration with passthrough devices is difficult (see Section 2.3.3) and not yet supported in vendor-supplied drivers or COTS hardware, and still subject to active research [101, 102, 103, 104, 105, 106].

4.2.3.2 Attack Throttling via Scheduling

A second solution, which leaves the attacker VM operational on the original machine, but still mitigates the DoS attack, can be realized by throttling the attacker VM. By enforcing a schedule on the attacker that prevents it from producing harmful PCIe transaction rates over a longer period of time, congestion on the interconnect can be capped. The approach uses a scheduling algorithm that was inspired by the cpulimit tool [127]. It is depicted in Figure 4.4.

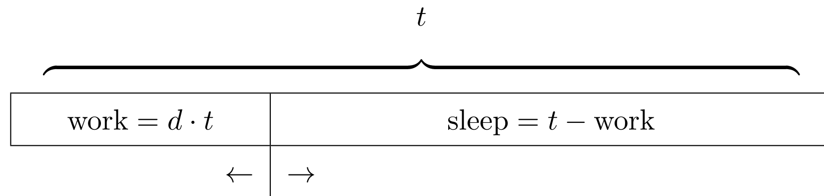


Figure 4.4: Throttling malicious VMs via scheduling. Parameter d is adjusted on-the-fly with help of monitoring feedback.

A scheduling timeslice for VMs with configurable duration t is divided into work and sleep quantums. The goal is to partition both quantums in a way that during t, the PCIe transaction rate of an attacker VM, R_{attacker}, stays below an allowed threshold transaction rate R_{allowed}. Partitioning of the quantums is done with help of the parameter d, which is dynamically adjusted during runtime. A first approximation of d can be calculated by using the monitors to count the number of PCIe transactions n_{counted} of the attacker VM during a timeslice with $d = 1$, and then recalculate d as

$$d = \frac{n_{\text{allowed}}}{n_{\text{counted}}}. \tag{4.1}$$

Here, n_{allowed} is the maximum allowed number of transactions during t, which can be calculated as

$$n_{\text{allowed}} = R_{\text{allowed}} \cdot t. \tag{4.2}$$

Afterwards, d can be dynamically fine-tuned by reading n_{counted} from the monitors again after each timeslice, and increasing or decreasing d accordingly in small steps. However, $d \in [0, 1]$ must always hold true. Mitigation results get better for small t, but this is capped by the scheduling resolution that the system's CPU and operating system can achieve.

4.3 Implementation

This section presents a proof-of-concept prototype implementation of the design ideas of the previous section. The prototype was developed on a cloud-grade x86 Xeon machine featuring the same 82576 dual-port Gigabit Ethernet NIC with SR-IOV support that was used in Chapter 3. As exploration results in the previous section showed, PCIe transaction monitoring lacks the needed granularity in current hardware. Therefore, monitoring and hardware-assisted DoS detection were implemented on a PCIe-capable Xilinx VC709 FPGA board, which then emulated these features for the 82576 NIC.

The decision to build a prototype that works as if monitoring takes place inside the I/O device was made due to the results of the exploration in Section 4.2.2. It showed that (1) existing COTS I/O devices already provide hardware facilities to detect malicious drivers and (2) DoS detection is less ambiguous inside the I/O device. Both reasons make I/O devices likely candidates for future incorporation of the presented concepts.

4.3.1 Overview

Figure 4.5 depicts a block diagram of the prototype. It used an Intel S2600COE dual-socket Motherboard that features a C602 chipset as the Platform-Controller-Hub (PCH). The first socket was populated with a 2.3 GHz Xeon E5-2630v1 six-core CPU. In order to minimize measurement variances and guarantee reproducible results without loss of generality, some precautions were taken. First, Hyperthreading was disabled, so that VMs could be pinned to physical cores without logically sharing them with other threads/VMs. Second, SpeedStep and TurboBoost technologies were disabled in the BIOS in order to prevent variations due to non-deterministic frequency scaling. Third, the second socket of the Motherboard was unpopulated. This prevented influencing the measurements due to non-uniform memory accesses (NUMA) and other latencies that emerge when a socket must be crossed.

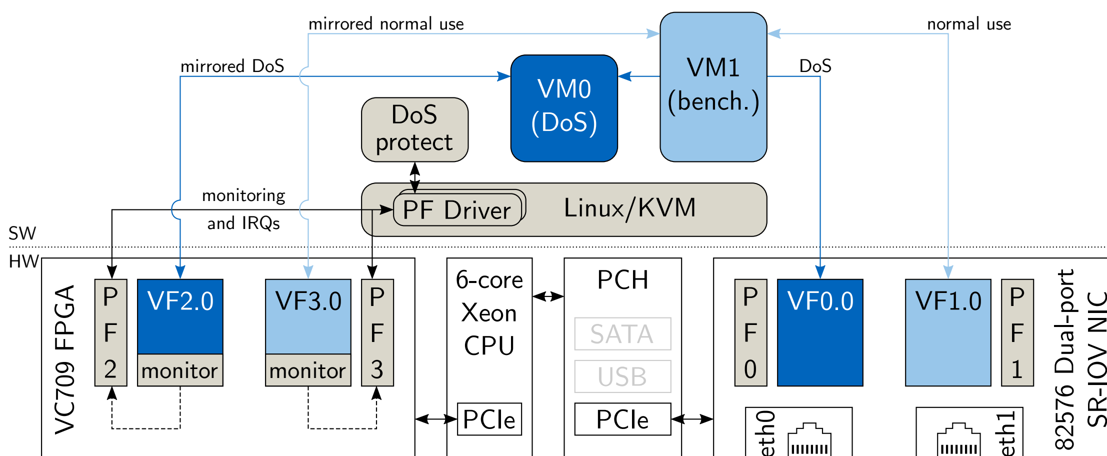


Figure 4.5: Block diagram of the prototype implementation. VMs run on dedicated cores.

The 82576 dual-port SR-IOV NIC was configured to spawn one VF for each PF; VF0.0 was associated to PF0, and likewise, VF1.0 to PF1. The prototype's CPU and PCH provided the necessary hardware extensions to use PCIe passthrough, so that both VFs could be directly attached to virtual machines. VF0.0 was attached to VM0 and was utilized as a target VF for DoS attacks. VF1.0, on the other hand, was attached to uncompromised VM1, which was used to conduct performance benchmarks that represented cloud computing workloads. In order to increase comparability to the experiments from the previous chapter, the 82576 was again connected to the system's PCH.

4.3.2 Monitoring and DoS Detection on FPGA

The 82576 NIC is a COTS product that has its controller integrated on an ASIC. Hence, it was not possible to retrospectively extend it with monitoring or DoS detection features. In order to solve this problem and build a prototype that worked as if the 82576 had this functionality, monitoring and DoS detection was implemented on an FPGA. Low overhead software extensions in the 82576 vendor driver then enabled the FPGA to monitor PCIe transactions that were sent to the 82576. These software aspects will be explained shortly in Section 4.3.4. First, the hardware implementation on the FPGA is presented.

The FPGA was part of a Xilinx VC709 evaluation board that was connected to the prototype through a CPU-integrated PCIe port. The board featured a Virtex-7 XC7VX690T FPGA that provided an integrated block for PCIe Generation 3 connectivity. The latter was used to realize a PCIe Endpoint on the FPGA. The integrated block also natively supported SR-IOV with up to two PFs and six VFs that could be allocated arbitrarily to the PFs. The PCIe Endpoint example design that came with the Xilinx Vivado 2013.3 suite was used as a starting point for the implementation of the monitoring and DoS detection functionality. The hardware description language was Verilog; A block diagram is depicted in Figure 4.6.

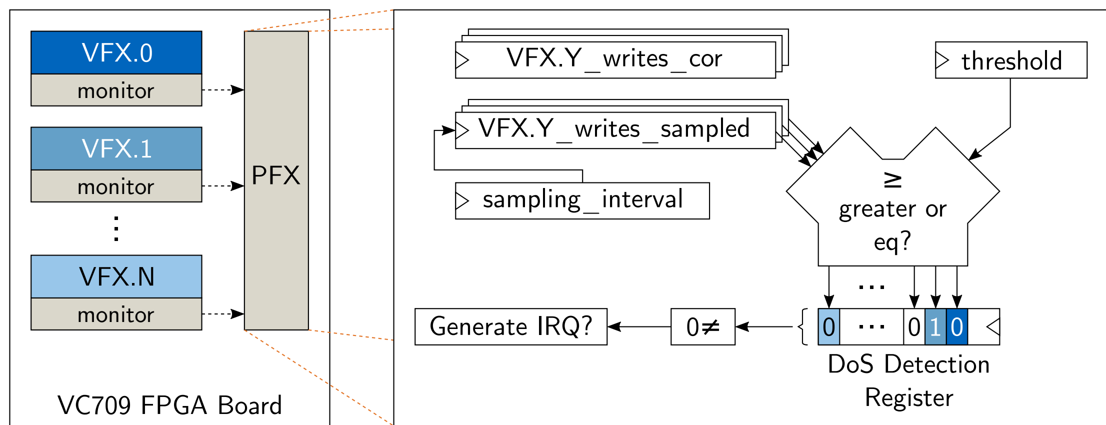


Figure 4.6: Implementation of hardware monitoring and DoS detection on the Virtex-7 FPGA. While the implementation is able to support a generic number of VFs, only a single VF per PF was spawned on the running prototype (compare Figure 4.5).

The block diagram shows one register set per VF that counts every incoming PCIe MMIO write transaction. The counters work in a clear-on-read fashion, which is indicated by `_cor` in Figure 4.6. This ensures low overhead read-out or sampling via host software. Each counter can be independently read from trusted host software through the PF to which the respective VF belongs. These counters are primarily used by host software to throttle an attacker VM, which will be presented in detail in Section 4.3.3.

The remaining blocks describe how hardware-assisted DoS detection on the FPGA works. Two parameters must be provided by the host for correct functionality: A sampling interval and a threshold value for PCIe transactions per sampling interval. Hardware then periodically compares each VF's sampled value (`VFX.Y_writes_sampled`) to the threshold, and if it is equal or greater to the threshold, set a bit in the DoS Detection Register. The latter is accessible to host software and uses the respective bit positions to encode which VF has been detected as an attacker. For instance, Figure 4.6 shows that bit position one stores a '1', which indicates that VFX.1 has been identified as a DoS attacker. After detection of a DoS attack, sampling is paused, which freezes the current state of the detection register, and a PF interrupt is sent to the host. Host software can then fetch the detection register's contents, identify the attacker VM and take appropriate measures. DoS detection in the FPGA is resumed by clearing the detection register from the host.

4.3.3 VC709 PF Driver and DoS Protect Process

Interfacing the VC709 monitoring and DoS detection facilities was done with a host kernel driver for the VC709 PFs (compare Figure 4.5). It was used to read the counter values of the respective VF monitors and to set the DoS detection parameters (threshold and sampling interval for transactions). Additionally, the driver was responsible for handling VC709 IRQs that were generated on DoS attack detection. IRQs were forwarded to a userspace process, which was ultimately responsible for mitigating DoS attacks. In Figure 4.5, it is named `DoS protect`.

Interrupt delivery was realized in a fashion similar to the Linux userspace I/O (UIO) model [128]. The PF driver generated a character device file that blocked on `read()` syscalls until a PF interrupt arrived. On interrupt arrival, the PF's interrupt service routine acknowledged the IRQ, fetched the contents of the DoS Detection Register (DDR) and returned its content to the `DoS protect` userspace process that was blocked on the `read()`. Writing to the same character device file cleared the DDR, which re-enabled interrupts from the VC709.

On interrupt arrival, `DoS protect` used the DDR contents returned by the `read()` syscall to determine the PCI(e) Bus:Device:Function (BDF) notation of the VF under DoS attack. This was realized via information exported from the kernel's sysfs. Afterwards, `DoS protect` searched for the process ID (PID) of the QEMU process to which the attacked VF is assigned. For the prototype, this task was realized by parsing the kernel's process tree, looking for QEMU processes and inspecting their command line arguments. One of the arguments contained the BDF notation of the attacked VF, which revealed the attacker QEMU process ID and therefore the attacker VM.

Knowing the PID, `DoS protect` could start freezing or throttling the attacker VM. For the latter, the algorithm presented in Section 4.2.3.2 was used with a timeslice of $t = 500\,\mu s$. Recalculation of the work/sleep ratio after each timeslice was realized with the counter values of the clear-on-read registers. `DoS protect` could also back off and return the VM to normal system scheduling if no more flooding was detected for a certain period of time.

During active throttling of an attacker VM, the CPU overhead of `DoS protect` was measured at 5%. In systems with static pinning of VMs to CPU cores, `DoS protect` can be dynamically pinned to the same core than the attacker VM, which prevents stealing CPU cycles from concurrent VMs. If there are no DoS attacks in the system, `DoS protect` sleeps while waiting for an interrupt from the VC709, and therefore does not produce any CPU overhead at all.

4.3.4 System Software and Emulation of Monitoring in the 82576

For system software, Ubuntu 14.04 with a Linux 3.13 kernel was used. QEMU [44] and the KVM [43] hypervisor were used for virtualization. Two fully virtualized Linux guest VMs were spawned and ran the same OS and kernel as the host. Each VM was assigned 4 GB of RAM (total 32 GB) and pinned to its own dedicated core. Like mentioned in Section 4.3.1, each VM was attached to one VF of the 82576 NIC (VM0 $\rightarrow$ VF0.0, VM1 $\rightarrow$ VF1.0). Vendor drivers for the 82576 VFs were already provided by the 13.3 Linux kernel under the name of `igbvf`, so that the VFs could be used out of the box.

Small software modifications in the `igbvf` driver and the DoS attack code finally enabled emulation of a system where the transaction monitoring and DoS detection facilities of the VC709 were used to monitor MMIO transactions that were sent to the 82576. This was realized in the following way:

1. The VC709 board was configured to “clone” the 82576 configuration, so that there was one VF per PF. VF2.0 was spawned by PF2, and VF3.0 was spawned by PF3.

2. The VFs were attached to VM0 and VM1 respectively, so that VM0 had concurrent access to VF0.0 of the 82576 and to VF2.0 of the VC709. Likewise, VM1 had access to VF1.0 (82576) and VF3.0 (VC709).

3. The original `igbvf` driver was modified so that each instruction that generated a PCIe MMIO write transaction to a 82576 VF was executed twice. The second time, however, the PCIe transaction was sent to the complementary VC709 VF instead of the 82576 VF. The transactions were basically mirrored.

4. The same mirroring was also implemented for the DoS attack code that was executed in VM1.

This mirroring concept is also depicted in the block diagram in Figure 4.5. Most transactions could be conveniently mirrored by modifying the `regs.h` file of the `igbvf` driver, because it defined macros for writing to the 82576 MMIO registers. These macros

were almost exclusively used in the remaining source files of the driver. In the few cases where the macros were not used, mirroring was inserted manually. Modifications to `regs.h` are shown in Listing 4.1.

```
1  /* Define macros for handling registers */
2  extern void __iomem *vc709_vf_bar0_addr;
3
4  ...
5
6  //#define ew32(reg, val)    writel((val), hw->hw_addr + E1000_##reg)
7  static inline void ew32_mirror(volatile void __iomem *reg, u32 value)
8  {
9      writel(value, vc709_vf_bar0_addr);
10     writel(value, reg);
11 }
12 #define ew32(reg, val) ew32_mirror(hw->hw_addr + E1000_##reg, (val))
13
14 ...
15
16 //#define array_ew32(reg, offset, val) \
17 //        writel((val), hw->hw_addr + E1000_##reg + (offset << 2))
18 #define array_ew32(reg, offset, val) \
19     ew32_mirror(hw->hw_addr + E1000_##reg + (offset << 2), (val))
```

Listing 4.1: Excerpt of small modifications to the `regs.h` file of the Linux `igbvf` driver that enable mirroring transactions of 82576 VFs to VC709 VFs. Original macros are commented out for comparison.

Obviously, executing each PCIe transaction in the NIC driver twice imposed a small overhead on the benchmarks and measurements that will be presented shortly in the evaluation in Section 4.4.7. However, the overhead turned out to be between zero and a single-digit percentage, and was therefore suitable for this proof-of-concept implementation. Detailed numbers will also be presented in the evaluation. Additionally, it must be kept in mind that there would be no software overhead at all if monitoring and DoS detection facilities are implemented into the passthrough I/O device itself (in this case, the 82576 NIC), like it is proposed in Section 4.2.2.2, and not emulated via an external FPGA like in this prototype.

4.4 Evaluation

In this section, viability of the design and its concepts is demonstrated by presenting evaluation results of the prototype implementation. Therefore, the prototype was first used to determine an appropriate, real-world threshold value for PCIe transaction rates that differentiate DoS attacks from normal workloads. Subsequently, results for both mitigation approaches (throttling an attacker via scheduling or instantly freezing it) are presented. The evaluation was done by employing three established networking micro- and macro-benchmarks that represent typical cloud computing workloads. They are introduced in the following.

4.4.1 Benchmarks

Memcached [129] is an open source, high-performance and distributed memory object caching system. It is an in-memory key-value store that is used to accelerate dynamic web applications of all sorts by caching frequently used database requests or small static page elements. Memcached was executed in VM1 (compare Figure 4.5) and its performance was measured by loading it with the memaslap benchmark. The latter was running on a dedicated remote machine. Memaslap was configured to run with four threads and a concurrency value of 64. All other parameters were set to default values, so that requests were randomly distributed with probabilities of 90% for get and 10% for set requests. The performance metric reported by memaslap is transactions per second.

Apache is an established HTTP server and therefore represents a classic cloud application. It was also running in VM1 and served four static HTML pages that were 4 KiB, 16 KiB, 64 KiB and 256 KiB in size. A remote machine was running the ApacheBench (ab) benchmark with four concurrent threads and measured the number of completed page requests per second.

netperf [121] was again used to measure TCP and UDP streaming throughputs, like it was done throughout Chapter 3. Here, VM1 was initiating the streams to a remote machine; Message sizes were varied between 16 B and 4 KiB. This range ensured that the benchmark was executed in both, CPU-bound and I/O-bound conditions.

The remote machine that was used to conduct all three benchmarks was equipped with a Core i7-3770T CPU, which provided four physical cores. Furthermore, the machine was configured with 16 GB of RAM and an on-board Gigabit Ethernet NIC. In terms of system software, it had the same Ubuntu version installed that was used on the prototype's host and VMs. Throughout all benchmarks, a default value of 1500 was used for the Maximum Transmission Unit (MTU), and no jumbo frames were used. The benchmarks ran for 10 seconds and all presented results depict an average value from five benchmark runs.

4.4.2 DoS Attack Parameters

Evaluations in this section used DoS attack parameters that were different from those that were employed in Chapter 3. In contrast, DoS attacks here created 64 bit instead of 32 bit PCIe packets, and addressed the Receive Descriptor Base Address Low (RDBAL) register of the respective 82576 VF, instead of flooding the first register of its MMIO address range. These parameters were chosen because they achieved the most performance degradation for the given NIC, so that the mitigation approaches could be evaluated against a worst case DoS attack.

The choice of parameters was different because at the time the experiments of Chapter 3 were conducted and published, there was no awareness about these parameters and their influence on performance degradation. Hence, results in this chapter show worse

degradation than results of the previous chapter. Additionally, the overall change in degradation was also influenced by hardware differences to the previous chapter's platform. For example, CPU and PCH were exchanged to a Xeon machine in order to better resemble a cloud computing machine. A first idea about the employed DoS attack impact is given by Table 4.1, which compares latencies for reading from an attacked 82576 NIC VF during a DoS attack and while the system is idle.

Table 4.1: Latencies for reading from a 82576 NIC VF with and without concurrent DoS attack on VF0.0.

		82576 Latency
Concurrent DoS attack?	✗	1.63 µs (±16 ns)
	✓	18.82 µs (±1.9 µs)
Increase		1054,6 %

4.4.3 DoS Detection Threshold

The PCIe transaction rate threshold for detecting DoS attacks always depends on the specific hardware composition of the respective system. The threshold is defined/constrained by parameters like CPU speed, PCIe link bandwidth and processing time for PCIe packets, and must therefore be evaluated experimentally on the respective system. Evaluation was done by recording transaction rates for all three benchmarks and their respective parameter ranges that were presented in Section 4.4.1. All the benchmarks were executed in VM1. Additionally, the PCIe transaction rate for a DoS attack was recorded, which was executed from VM0. Results of the evaluation are depicted in Figure 4.7.

Results for the netperf streaming tests and Apache only show the result for the parameter choice that resulted in the highest transaction rate. This was a 128 B message size for TCP streams, a 16 B message size for UDP streams and 4 KiB page size for the Apache server.

The bottom line of the results is that the PCIe transaction rate of a DoS attacker VM is round about five times larger than the maximum rate for the most demanding legal benchmark (netperf UDP in this case). This large gap in transaction rates between legal usage and DoS attack made the definition of a threshold value a comfortable task. All in all, there are three approaches to define a threshold: (1) Picking a rate just above the highest legal benchmark, (2) picking a rate just beneath the recorded DoS attack rate, (3) or any value in between. For this evaluation, option one with a transaction rate threshold of 420 000 wr/s was chosen. This was about 1% above the netperf UDP results and did not trigger any false positives throughout the evaluations. After configuring the VC709 DoS detection with this threshold value, the system was used to evaluate both mitigation approaches (attacker throttling vs. freezing) with help of the three benchmarks. In the following, benchmark results are presented for mitigating attacks by

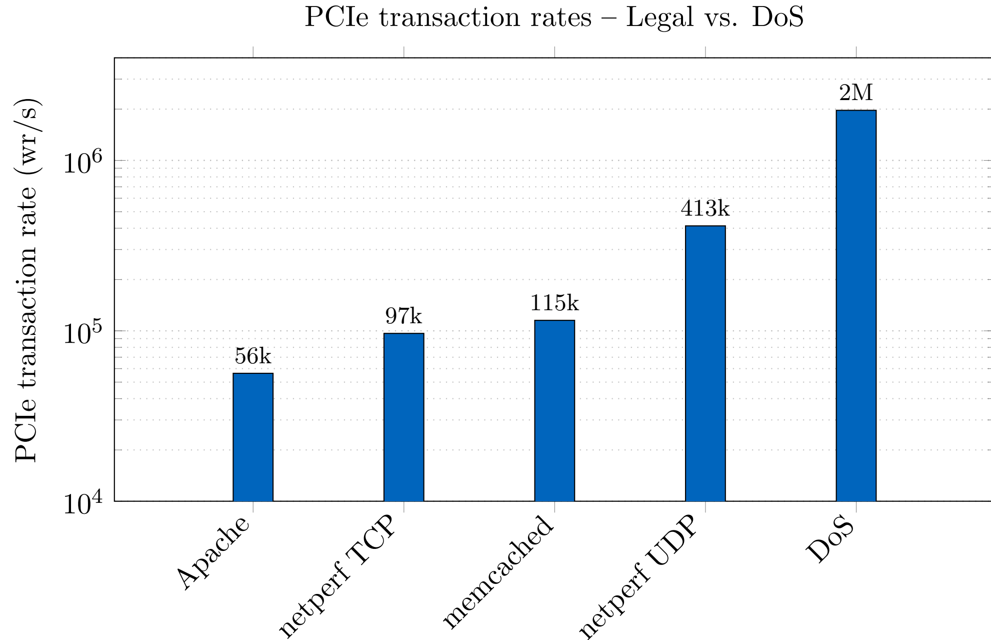


Figure 4.7: Maximum CPU-to-VF PCIe transaction rates for multiple cloud-relevant benchmarks and a DoS attack. Depicted netperf TCP used 128 B message size, netperf UDP 16 B. Apache used 4 KiB page size.

throttling attacker VMs via scheduling, as presented in Section 4.3.3. Therefore, results show baseline performance, performance during a DoS attack without any mitigation, and performance during a DoS attack with active mitigation (aka throttling). Mitigating attacks by freezing an attacker can be compared to restoring baseline performance. A trace of this approach will be presented afterwards in Section 4.4.6.

4.4.4 Memcached and Apache Results

Figure 4.8 depicts results for the memcached benchmark. A DoS attack had great impact on performance by reducing transactions per second from 88671 to 54784, which is a decline by 38,2%, or 33887 transactions in absolute numbers. Detecting and throttling the attacker VM restores performance to 88147 transactions per second, or 99.4% of baseline performance.

Figure 4.9 shows results for the Apache benchmark and the four different static page sizes. During a DoS attack, page requests per second dropped by 8.4% (446 req/s) for 4 KiB pages. With increasing page sizes, the performance degradation gets worse. For 16 KiB pages, performance drops by 33.5% or 1386 req/s. Page sizes of 64 KiB and 256 KiB sizes see degradations of 46,1% (725 req/s) and 51,3% (229 req/s), respectively.

Throttling the DoS attacker VM made performance return to almost baseline numbers. The page request rates returned to values between 97% (16 KiB) and 99% (256 KiB) of baseline performance.

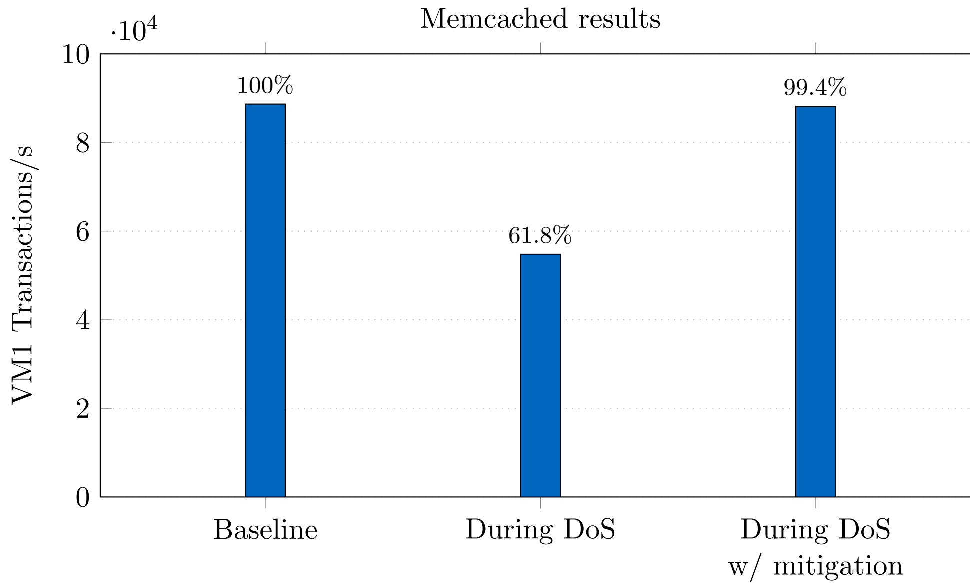


Figure 4.8: Results for the memcached benchmark. Mitigating the DoS attack by throttling the attacker VM restores 99.4% of baseline performance.

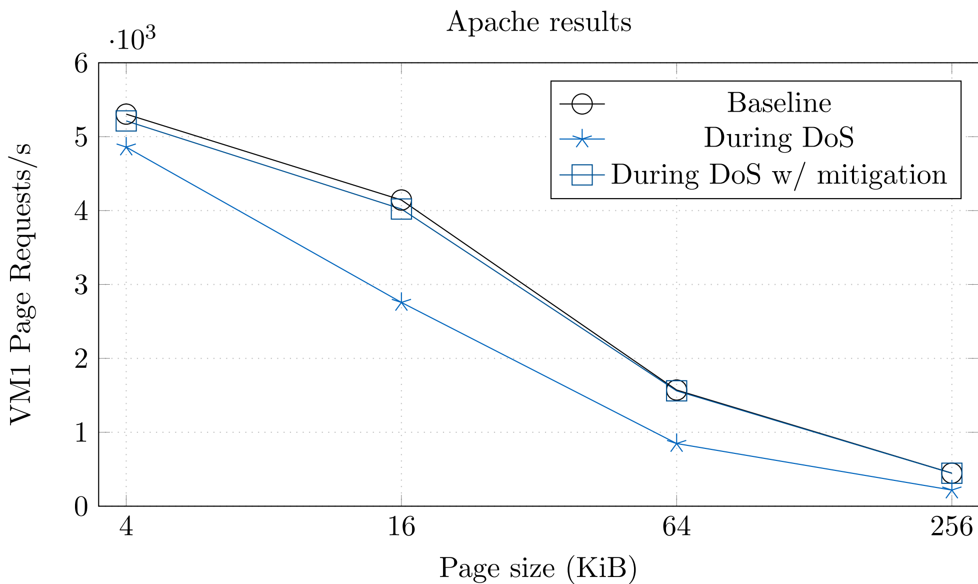


Figure 4.9: Apache results for four page sizes. Mitigation restores between between 97% (16 KiB) and 99% (256 KiB) of baseline performance. The x axis has logarithmic scaling.

4.4.5 Netperf Results

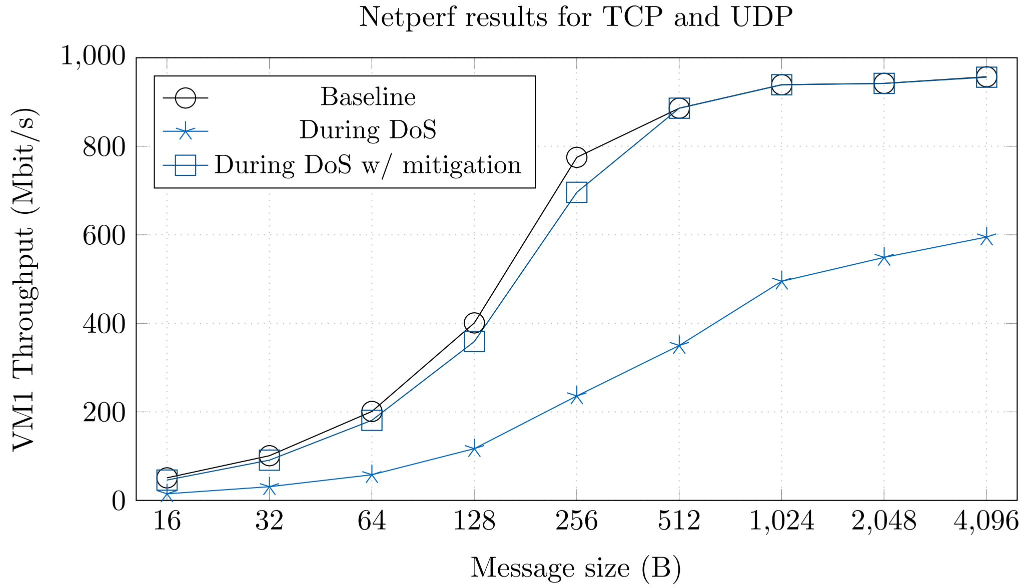
Figure 4.10 depicts results for netperf TCP and UDP streaming benchmarks and the respective message sizes. In summary, both protocols see performance degradation during DoS attacks if mitigation was not activated, but the actual degradation depended on message size. For UDP streams, performance degradations between 37.8% (4096 B message size) and round about 70% (256 B - 16 B message size) were observed. Mitigating the DoS attack by throttling the attacker VM restored baseline performance for message sizes between 4096 B - 512 B. For the remaining range of message sizes (256 B - 16 B), performance was restored to about 90% of baseline performance by the throttling approach.

Results for streams using the TCP protocol showed similar behavior. Performance degradation during DoS attacks for the message size range of 4096 B - 256 B was measured at 51%. For the same range, throttling the attacker resulted in a return to full baseline performance. For the remaining range of message sizes, performance degradation during DoS attacks decreased with smaller message size. At 128 B, a 47% degradation was measured, while it was only 1% for 16 B message sizes. In the same range, mitigation via throttling restored between 86.5% (128 B) and 93.7% (16 B) of baseline performance.

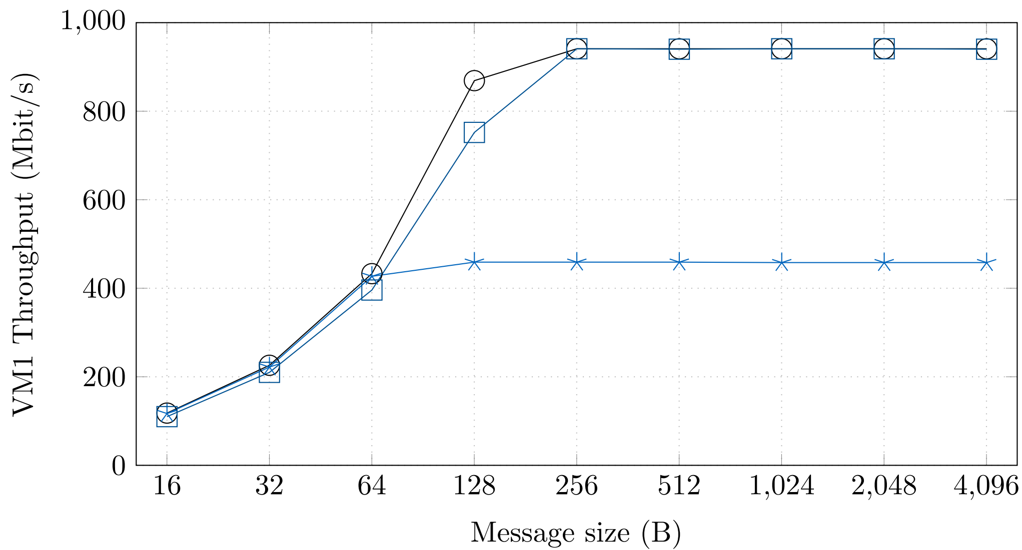
The results for both protocols, UDP and TCP, showed that mitigation of DoS attacks by throttling the attacker VM yielded different results depending on the message size of the netperf streams. In conclusion, baseline performance could be restored for all message sizes where the netperf benchmark was not CPU-bound. This was the case for message sizes of 512 B and greater for UDP streams and 256 B or greater for TCP streams.

For respective smaller message sizes, DoS attacks were still able to cause some degradation despite being actively throttled. As described in Section 4.2.3.2, throttling means that attacker VMs are scheduled only for very small time slices. The size of a time slice, however, is bounded by CPU speed and the performance of the host operating system's scheduling code. Evaluations found that even for the smallest possible time slices, attacker VMs were still able to cause a short lasting congestion on the PCIe interconnect. These short lasting congestions had no impact if netperf was I/O-bound, aka operated with big message sizes and the CPU needed to wait for the I/O device most of the time. However, if the netperf benchmark was CPU-bound, which means that a VM's CPU core was fully loaded because it needed to prepare large numbers of small UDP or TCP packets during small amounts of time, the mini congestions were still enough to degrade some performance. Performance, in this case, was lost when a fully loaded CPU core, which executed the legal netperf benchmark, needed to wait for a few additional cycles in order to send a PCIe packet downstream, because another CPU core, running the DoS attack, caused a mini congestion on the interconnect that delayed packet sendout.

However, the throttling approach worked fine for both macro-benchmarks (Memcached and Apache) that represented real-world workloads, and therefore the requirement for sufficient mitigation was satisfied. Further optimizations of the scheduling algorithm were therefore not investigated.



(a) netperf UDP



(b) netperf TCP

Figure 4.10: Netperf throughput results for UDP and TCP streaming benchmarks. The x axes have logarithmic scaling.

4.4.6 Trace of DoS Detection and Attacker Freeze

In the previous three sections, results were presented for mitigating DoS attacks by throttling the attacker VM with a scheduling algorithm. A second mitigation approach, which was presented in Section 4.2.3 as an alternative to throttling, is to instantly freeze an attacker VM (and optionally migrate it to an isolated host for further inspection). In terms of performance, this approach can be compared to restoring baseline performance. Freezing the attacker means that there will be only a very short, one-time congestion on the interconnect. The congestion duration is determined by the time the system needs to detect the DoS attack, determine the attacker VM and finally freeze it. In order to demonstrate this approach, a trace of the system detecting and freezing a DoS attacker is presented in the following.

Therefore, the DoS detection of the prototype was configured the same as presented in Section 4.4.3, which means it checked for a transaction rate of over 420 000 wr/s per VF. This was realized by programming the detection hardware with a sampling interval of 0.2 s and a transaction threshold of 84 000 wr (compare Section 4.3.2). The trace covered 10 s containing the following events:

1. At the start of the trace, the whole system was idle.
2. At second two, VM1 started a netperf benchmark using the UDP protocol and 4096 B message sizes.
3. At second six, VM0 started executing a DoS attack on its assigned VF0.0. DoS attack parameters also were the same as in the previous sections.

To give insight into the state of the system and the workings of DoS detection and mitigation by freezing the attacker, three parameters were recorded during the trace. These are (1) the network throughput that VM1's netperf benchmark achieved, (2) the PCIe transaction rate of both VMs and (3) the latency of the 82576 NIC. In order to show the difference between the system with and without the mitigation approach, the trace was run two times. The first run represented a baseline run, where DoS detection and mitigation was disabled. For the second run, it was activated and contained a successful mitigation. The trace results are depicted in Figure 4.11.

4.4.6.1 Baseline Trace

The left column depicts the baseline run and shows the impact of a DoS attack on the system with deactivated mitigation. The top row depicts the netperf throughput of VM1 during the trace, the middle row shows the PCIe transaction rate of both VMs and the bottom row depicts the latency for reading a data word from the (attacked) 82576 NIC. For scaling reasons, the transaction rate plots show a second ordinate for the PCIe transaction rate of the DoS attacker VM0. The netperf benchmark is observable via the throughput, the PCIe transaction rate of VM1 and the minor jitter it causes to the latency of the 82576 NIC. The DoS attack, in turn, is visible via VM2's PCIe transaction rate and its impact on the other plots. Once the DoS attack started at second six of

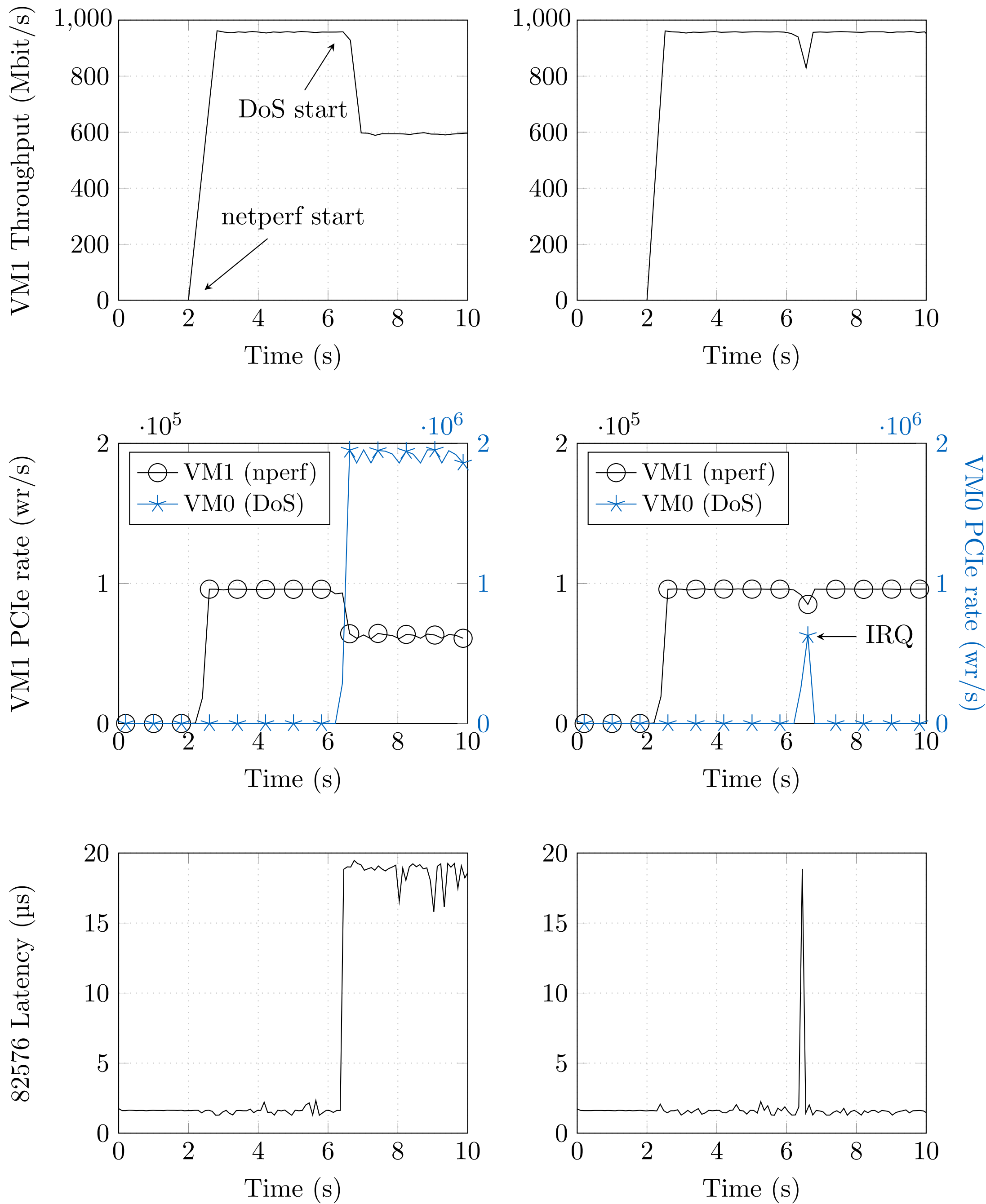


Figure 4.11: A trace of 10 seconds that demonstrates freezing a DoS attacker. Left column shows the impact of a DoS attack (visible as the PCIe transaction rate of VM0) on a netperf benchmark (running in VM1) with deactivated mitigation. Right column has it enabled. Please note the different ordinates for the PCIe rate plots.

the trace, netperf throughput instantly dropped from 960 Mbit/s to 600 Mbit/s. This performance degradation correlates to a reduced PCIe transaction rate of VM1. The cause for this degradation was the DoS-caused congestion on the interconnect, which was discussed in Chapter 3. The congestion is made visible by the latency plot, which shows a steep jump once the DoS attack was executing[1].

4.4.6.2 Mitigation Trace

The right column shows results for enabled mitigation. Here, the VC709's DoS detection identified the attack after it was started and reported VM0 via interrupt. The DoS protect process in the host received the interrupt shortly after and instantly froze VM0. As a result, the DoS attack in this trace only caused a small dip in VM1's throughput and performance was recovered to 100% of baseline shortly after the attacker was identified and frozen.

4.4.6.3 Optimization Tradeoffs

It is also possible to detect DoS attacks faster than in the depicted trace, which would have further reduced the temporary performance drop. Therefore, other parameters must be chosen for the sampling rate and the transaction threshold of the DoS detection hardware in the FPGA. The FPGA design runs at 125 MHz, which theoretically enables it to check for the transaction threshold value at smaller sampling rates than the 0.2 s that were used, allowing faster detection and freezing of the attacker. On the other hand, greater sampling rates provide more confident detection. In cloud computing environments, the latter is most likely more important, because while a tiny performance drop is annoying, it is still preferable to having more false alarms.

4.4.7 Mirroring Overhead

In the following, the overhead that transaction mirroring introduced to the conducted evaluations and benchmarks will be discussed. Of course, all findings within this section need only be considered for the prototype implementation, because transaction mirroring was only used to enable evaluation of the original design idea. If respective monitoring hardware would be implemented directly into hardware, like it is proposed by the design, no mirroring would be needed and therefore no overhead would be introduced.

Transaction mirroring causes CPU cores to duplicate each PCIe transaction that is sent to the 82576 NIC. This translates to (1) a tiny CPU overhead for generating the additional PCIe transaction and (2) a doubling of the volume of PCIe transactions. The latter is neglectable due to two reasons. First, the prototype (compare Figure 4.5) is assembled such that the VC709 FPGA and the 82576 NIC PCIe cards do not share any PCIe lanes. Therefore, mirroring introduced no interferences in this part of the PCIe subsystem. Second, the CPU internal ring structure, which must be traversed by the

[1] Compare to results that were presented in Section 3.5, where huge latency increases were identified as an indicator of DoS-caused congestion on the interconnect.

original and the mirrored PCIe packets, is capable of much larger transactions volumes than those introduced by two PCIe devices.

The impact of the CPU overhead introduced by the mirroring approach is discussed in the following.

4.4.7.1 Benchmark Overhead

VM1, which executed the benchmarks, had transaction mirroring enabled only for the evaluation of the maximum PCIe transaction rates per benchmark in Section 4.4.3, and the system trace for evaluating the freezing approach that was presented in Section 4.4.6. Here, the CPU overhead of mirroring introduced variations in the benchmark results of at most 7%, but only when the benchmarks itself was CPU-bound. Therefore, in both cases, this overhead was neglectable. In the first case, mirroring was used to determine the difference in PCIe transaction rates between DoS attacks and legal benchmarks. As the DoS rate was round about five times larger than the next best legal benchmark, a 7% overhead for the latter was acceptable. Considering the results of the system trace in Section 4.4.6, netperf was not operated in the CPU-bound region (using 4096 B message sizes), so that CPU overhead of mirroring had no effect. All benchmark results in Sections 4.4.4 and 4.4.5 were obtained without active mirroring in the benchmark VM and thus contained no overhead.

4.4.7.2 DoS Attack Overhead

The DoS attack code had transaction mirroring enabled during the whole evaluation. However, no overhead was introduced because the VC709 FPGA was able to process PCIe transactions faster than the 82576 NIC. This is shown in Table 4.1.

Table 4.2: Times for executing a for-loop (DoS attack) with 10^8 consecutive PCIe write transactions, depending on device.

Device	total time	time/write
82576 NIC	36.138 s	361.38 ns
VC709 FPGA	8.812 s	88.12 ns

While the 82576 NIC needs 361 ns to process a single PCIe packet, the VC709 only needs 88 ns. Due to this difference in processing speeds, DoS-caused congestion on the PCIe subsystem first emerged on the PCIe lanes that connected to the 82576 NIC. This means that during congestion, when all buffers were filled, the CPU core running the mirrored DoS attack code would block for 361 ns while waiting for the 82576 NIC to process a PCIe transaction that frees a downstream buffer slot. Hence, this waiting time that existed anyways could be used by the CPU to send the respective mirrored packet to the VC709 without generating any overhead.

4.5 Summary

The cloud computing domain is the most prominent adopter of SR-IOV and PCIe passthrough technology. Cloud computing providers like Amazon use the technology to provide VMs that offer high performance networking I/O that is superior to legacy I/O virtualization options like paravirtualization. As a consequence, servers that host SR-IOV VMs are vulnerable to PCIe DoS attacks and may therefore become victims of unwanted, significant performance degradation (attacks and degradation were thoroughly investigated in Chapter 3).

This chapter presented research on detecting DoS attacks on PCIe passthrough hardware in live systems and mitigating the resulting performance degradation. The overall goal was to develop lightweight hardware and software extensions that work together in order to fulfill these tasks. In a first step, domain-specific goals and requirements were collected. The design of the hardware and software extensions should seek synergies with existing technology that counteracts other classes of VM misbehavior (e.g network address spoofing) and be compatible to existing and future hardware architectures and I/O classes. Most importantly, the design must keep the host out of the VM-to-I/O-device data path, which is the most important paradigm of PCIe passthrough. Also, the design had to satisfy requirements for non-intrusive, low overhead and secure DoS attacker detection. Additionally, attackers must be identified unambiguously, their attacks must be sufficiently mitigated and the overall design must scale to modern cloud computing requirements.

In the next step, these goals and requirements were used to develop a high-level design for the hardware and software extensions. The design was based on the observation that DoS attacks always create larger PCIe transaction rates than legal (cloud computing) workloads. Therefore, the design was partitioned to utilize hardware-based monitoring of PCIe transaction rates for DoS detection and host software extensions for mitigation. Exploration of hardware monitoring possibilities in PCIe networks yielded three results: (1) Monitoring inside the CPU, (2) in an intermediary like a switch, or (3) inside the I/O device. Each possibility provided its unique pros and cons, but all three would be suitable for implementation in future hardware. In terms of software extensions for DoS attack mitigation, two approaches were presented: (1) A scheduling algorithm that throttles the attacker VM so that the DoS attack weakens significantly, or (2) freezing and subsequently migrating the attacker VM to an isolated host, where it can be inspected by an operator of the cloud computing provider.

In order to evaluate the high level design, it was implemented on cloud computing grade, real hardware. An FPGA prototyping board was utilized to emulate hardware-based PCIe transaction monitoring and DoS detection for a commercial-off-the-shelf SR-IOV NIC. Using this hardware prototype, a threshold value for PCIe transactions rates was determined that differentiates legal usage of an SR-IOV VF from a DoS attack.

Finally, this threshold value and the hardware prototype were used to evaluate the DoS detection and software-based mitigation of DoS attacks in the live system. This task was realized by using three cloud computing benchmarks (one micro- and two macrobenchmarks). Results of the macro-benchmarks showed that the throttling algorithm

successfully mitigates performance degradation due to DoS attacks. For example, the memcached benchmark saw a drop in performance to 61.8% of baseline performance during a DoS attack. Throttling the DoS attacker with the proposed scheduling algorithm restored 99.4% of baseline performance for memcached. Results for the approach of freezing DoS attacker VMs showed that DoS detection is fast enough to only leave a sub-second dip in network performance until full baseline performance is restored.

5 Resolving Performance Isolation Issues Through PCIe QoS Extensions

The previous chapter presented approaches for mitigating performance isolation issues with PCIe passthrough in a running system. Mitigation approaches have a low footprint regarding costs, because they can be enabled with very small monitoring add-ons to existing hardware. These monitors detect PCIe DoS attacks, which are then mitigated by software. Although mitigation is able to restore close to baseline system performance, there still might be small temporary performance fluctuations, which are caused by delays in DoS attack detection and the effectiveness of the mitigation software. These characteristics make mitigation approaches well-suited for the cloud and datacenter computing domains. Here, detection of attackers is the most important aspect, because an operator (or software watchdog) of the cloud computing provider can shut down or fix detected attacker VMs from remote. In the short meantime between DoS attack detection and reaction from the operator, it is perfectly acceptable to sacrifice a few percent of performance and therefore save money due to an affordable and lightweight implementation. In other words, DoS detection and mitigation are a reactive solution, because it counters on-going attacks.

There are also other computing domains, where a reactive solution is not the preferred way. For example, in future embedded and mixed-criticality systems that use PCIe passthrough, performance fluctuations during DoS attacks and DoS mitigation would need to be accounted for in worst case execution time (WCET) calculations. Considering the many hardware and software parameters that influence these performance fluctuations, and the significant increase of latencies that is caused by DoS attacks, this would result in a very pessimistic approach that also introduces a high element of uncertainty. Even if these fluctuations would be accounted for in WCET calculations, it is a lot more difficult to fix a compromised VM in embedded systems than in cloud systems, because instant (remote-) maintenance by experts/operators might not be possible. Hence, in this domain, extra costs regarding hardware would be acceptable if they completely close this attack vector and enable usage of PCIe passthrough hardware without the need to calculate with any kind of interference. In other words, a proactive solution is preferred.

This chapter presents two integrated hardware architectures that achieve this goal. Therefore, Section 5.1 first defines overall goals and domain-specific requirements for the hardware architectures. As a foundation for both architectures, Quality-of-Service (QoS) extensions from the PCIe specification [78] were employed. The extensions are thoroughly introduced in Section 5.2. Unfortunately, at the time of writing, these QoS extensions were almost completely absent in COTS PCIe hardware, because they are

defined optional in the specification and lack use cases in the real world. In order to enable development and evaluation of the proposed hardware architectures despite the absence of PCIe QoS hardware, a SystemC TLM 2.0 simulation model of a real-world lab-setup was developed and extended with these PCIe QoS features. The model is presented in Section 5.3. Afterwards, Sections 5.4 and 5.5 present the two proactive hardware architectures for DoS prevention. They were optimized for different goals: Scheduling freedom and minimal hardware costs. In both sections, it is determined which QoS features are needed, and how virtualized hardware like multi-core CPUs and SR-IOV devices need to implement, interface[1]and program these features in order to prevent DoS attacks. Afterwards, evaluation results of each architecture using the SystemC model are presented and a short summary for each respective architecture is given. Finally, this chapter is concluded in Section 5.6.

5.1 Goals and Requirements

In order to enable hardware-assisted full performance isolation for systems using PCIe passthrough, certain goals and requirements must be met and satisfied. Both are formulated in the following. The goals and requirements are agnostic of specific multi-core processor architectures, PCIe intermediaries or I/O devices, so that they are applicable to any kind of common hardware as long as it implements standardized PCIe interfaces.

5.1.1 Goals

The overall goal is to develop hardware architectures that enable proactive, full performance isolation for PCIe passthrough. This goal shall not be achieved with any kind of help by software, e.g. introspection or monitoring mechanisms from a trusted host, but enforced solely by hardware. The only allowed exception shall be configuration of hardware via trusted host software at boot time, prior to VM instantiation. Additionally, the hardware architectures shall stay fully compatible to the PCIe and SR-IOV specifications. This ensures compatibility to existing legacy hardware, and simplifies integration of the developed solutions into future hardware, because they build on top of a common base.

5.1.2 Requirements

Proactive, full isolation: The hardware architectures must provide full performance isolation, which means that there must be zero I/O performance degradation for non-attacker VMs during DoS attacks. Also, isolation must be guaranteed permanently. It cannot be activated with a short delay after a DoS attack emerged, like in the DoS detection and mitigation approach in the previous chapter, but must be there from the start.

[1]These are aspects that are explicitly not covered in the PCIe specification.

Secure configuration: Any configuration facilities of the hardware architectures must be secured from access by VMs. Only trusted host software, e.g. a hypervisor, is allowed to access and alter the respective configuration.

Scalability: The hardware architectures must be scalable to a reasonable number of VMs and passthrough functions.

Low overhead: The hardware architectures must not introduce performance overhead to other, non-PCIe parts of the system.

5.2 Quality-of-Service Extensions in the PCIe Specification

Exploration of the PCIe specifications [78] found that they include modular hardware QoS extensions for generic PCIe components. Unfortunately, all of these QoS extensions are explicitly defined optional. At the date of writing, there seems to be a lack of use cases for QoS extensions in PCIe passthrough-capable hardware. Especially in the case of x86-based hardware, where PCIe passthrough found adoption in the form of SR-IOV based networking for virtualized cloud computing, the author is not aware of any COTS PCIe hardware[2] that provides QoS facilities as described in the PCIe specifications.

Performance isolation issues of PCIe passthrough and SR-IOV, however, qualify as a use case for these extensions. Closing the existing attack vector for PCIe DoS attacks with help of the specification's QoS extensions would have two advantages. First, compatibility to legacy PCIe hardware is increased, and second, integration into existing PCIe hardware is simplified. For these reasons, the QoS extensions were utilized as building blocks in the hardware architectures that are presented later in this chapter, and they help enabling full performance isolation for PCIe passthrough.

5.2.1 Contribution of this Thesis

Before this section continues with background on the specification's QoS extensions, it shall be first differentiated where this thesis leverages existing concepts from the PCIe specifications, where it transfers and adapts them to new use cases and where it contributes new work that is needed to enable full performance isolation.

PCIe's QoS extensions are included since the very first revision (1.0) of the PCI Express Base Specification from the year 2002 [130]. The first commercial IOMMUs for x86 hardware, which enabled PCIe passthrough, were introduced around the year 2006 [49]. The first revision of SR-IOV was introduced a year later in 2007 [131]. SR-IOV extended the PCIe base specifications, hence becoming a superset of them, but it did not alter, update or change specifications regarding the QoS extensions. Therefore, it is possible to design both legacy and SR-IOV capable PCIe devices that include or exclude QoS extensions. The whole relationship is depicted in Figure 5.1.

[2]For QoS to work, it must be supported pervasively by all components in the hierarchy of a PCIe tree. This results in at least one CPU and one Endpoint, if they are directly connected to each other. Any intermediaries between CPU and Endpoint, e.g. a chipset (PCH), must also provide matching QoS capabilities.

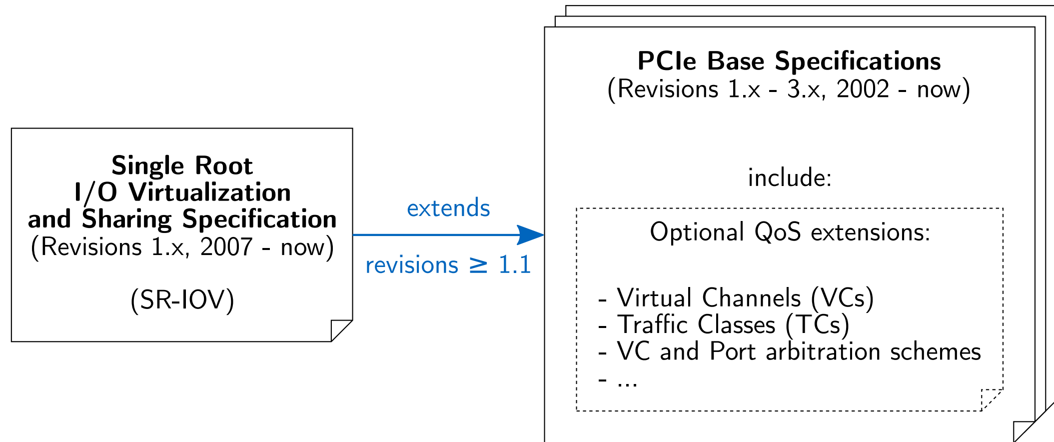


Figure 5.1: Relationship between the PCIe Base Specifications, the optional QoS extensions and SR-IOV. SR-IOV can be a superset of any of the PCIe Base Specifications that have a revision number of 1.1 or greater.

In conclusion, the QoS extensions were designed and released years before platform virtualization emerged as a mainstream technology, and DoS attacks from malicious VMs became an easily exploitable attack vector. Since the extensions are modular and highly configurable, work on this thesis included exploring which QoS components are needed and how they must be configured in order to achieve the goal of full performance isolation for the hardware architectures that are presented in Sections 5.4 and 5.5.

Although the specification includes a section on "isochronous mechanisms" [78, Appendix A] that are based on the QoS extensions, they could not be applied. The mechanisms enable two communication paradigms that provide guaranteed bandwidths and deterministic latencies for Endpoint-to-Root-Complex and Endpoint-to-Endpoint directions. PCIe DoS attacks from malicious VMs, however, flow downstream in the CPU-to-Endpoint direction. The specification excludes this direction for a second time in a section about multi-function devices (aka all SR-IOV devices), stating that "the multi-Function device model does not support full QoS management [...] for Downstream requests" [78, Chapter 6.3.3.4]. Hence, a new paradigm for preventing DoS attacks with help of the QoS extensions was developed for this thesis. Additionally, new concepts for interfacing QoS-enabled PCIe in modern virtualized hardware were developed. This was explicitly needed for virtualized multi-core CPUs, because here, the PCIe Base Specification says that "the mechanisms used by the Root Complex to interface PCI Express to the Host CPU(s) are outside the scope of this document" [78, Chapter 6.5.5].

5.2.2 Overview of the PCIe Specification's QoS Extensions

The specification's key feature is support for multiple Virtual Channels (VCs) per PCIe port. Together with Traffic Class (TC) labeling of PCIe packets and multiple arbitration stages and schemes, it is possible to enable different qualities of service. An example for a PCIe component with two ingress ports and two VCs is given in Figure 5.2.

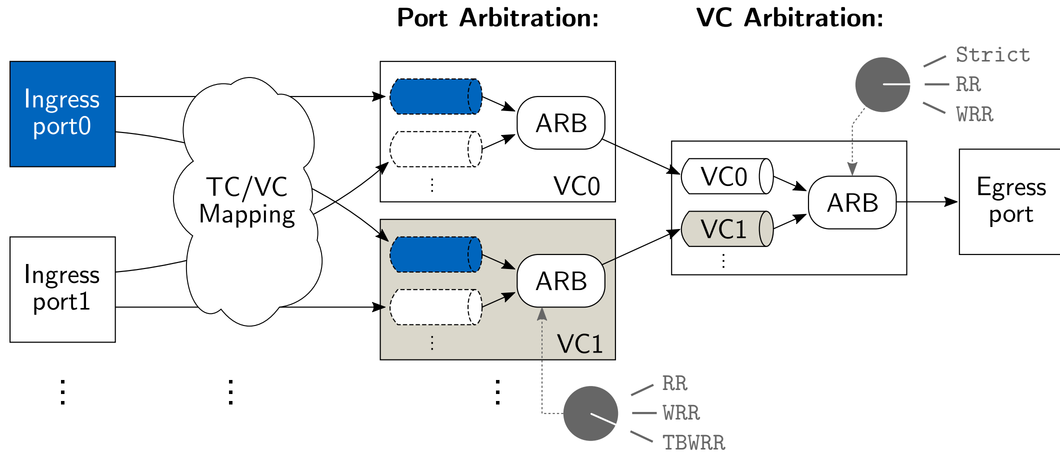


Figure 5.2: Overview of PCIe Quality-of-Service extensions. First, an inbound packet's Traffic Class label is used to map it to a Virtual Channel (VC). Afterwards, a two-stage arbitration (first port, then VC) selects the next packet.

5.2.2.1 TC/VC Mapping

VCs are dedicated hardware buffers that can be integrated for each individual PCIe port of a component. Only a single VC is mandatory, if more are employed, differentiated flows over that link are possible. Each VC of a port must provide its own flow control facilities. Association of traffic to individual VCs is realized by labeling PCIe packets with TC labels. Up to eight TCs (TC0–TC7) and VCs (VC0–VC7) are specified, and they must be mapped in a n-to-1 fashion, respectively. Association of the same TC label with multiple VCs is forbidden. Two examples for legal mappings are depicted in Table 5.1.

Table 5.1: Two examples for TC/VC mapping for a port with two VCs.

Mapping	VC ID	
	VC0	VC1
1st mapping	[TC0-TC3]	[TC4-TC7]
2nd mapping	[TC0-TC6]	[TC7]

In the first mapping, TCs are evenly divided between both VCs. The second example resembles a mapping that would be employed for separating a high-priority VC1, which is only used by the highest TC label, from a low-priority VC0, which is used by the remaining TCs. It is also mandatory that ports at both sides of a PCIe link are configured with identical TC/VC mappings.

5.2.2.2 Arbitration

After TC/VC mapping, a two-stage arbitration takes place. If a PCIe component has multiple ingress ports, then it is necessary to differentiate transactions that target the same VC but originate from different ingress ports. This is done in a first stage called **Port Arbitration**. Here, a different arbitration scheme can be selected for each individual VC (compare Figure 5.2). The following schemes are supported:

Hardware-fixed (RR): A simple, non-programmable scheme that gives equal priority to all ingress ports, for example Round-Robin (RR).

Programmable Weighted Round-Robin (WRR): Trusted host software, e.g. the hypervisor, generates an arbitration table that is stored in a read-write register array. The table entries contain port numbers of the respective ingress ports. At each arbitration phase, the current table entry is evaluated and the indicated ingress port is served for one transaction. If the respective port does not have a pending transaction, it is skipped and the arbiter immediately moves to the next table entry.

Programmable Time-based Weighted Round-Robin (TBWRR): Also operates with an arbitration table. Here, an indicated port is served for a single transaction during a fixed timeslot t, with $t = 100$ns in the current revision of the specification. The arbiter moves to the next table entry only after t has passed. If a port did not have a pending transaction during t, it is considered an idle phase. It is also possible to explicitly insert idle phases. Figure 5.3 depicts an exemplary arbitration table.

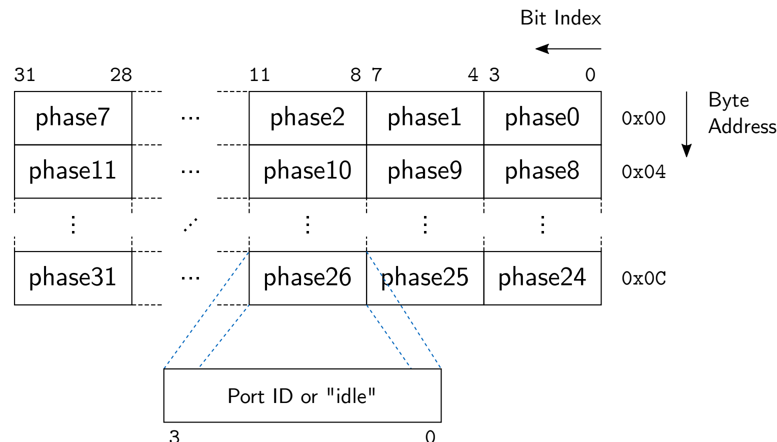


Figure 5.3: TBWRR arbitration table with 32 phases and 4 bit table entries. Per phase, it is possible to program arbitration of one of 16 ports or an idle phase.

After PCIe transactions passed the port arbitration stage, they are subject to the second arbitration stage, **VC Arbitration** (compare Figure 5.2). Here, VCs can be partitioned into two priority groups, a lower and upper group. Within the upper group,

a strict priority arbitration scheme is employed (VC7 > VC6 > ...). The lower group is considered for arbitration only if there are no pending transactions in the upper group. Within the low priority group, arbitration can be configured to use one of two schemes: (1) Round-Robin or (2) Programmable Weighted Round-Robin (WRR). The latter works just like the WRR scheme for port arbitration and also utilizes an arbitration table in a register array that is supplied by trusted host software.

5.3 SystemC TLM 2.0 Model of QoS-enabled PCIe Components

In order to enable design, exploration and evaluation of hardware architectures that employ PCIe's optional QoS extensions despite the previously highlighted lack thereof in COTS hardware, a simulation model was developed. To ensure that the model accurately simulates real-world systems that employ PCIe hardware, the following approach was taken beforehand:

1. Create a model of PCIe ports with configuration options for each of the optional QoS extensions.
2. Build models of configurable, generic system components that incorporate these PCIe ports: Multi-core CPU, chipset and (SR-IOV) I/O device.
3. Interconnect and configure the generic components to resemble a real-world lab-setup (no QoS activated in the simulated PCIe components yet).
4. Run a networking benchmark on the real-world machine, and replicate the benchmark on the simulation model. Compare benchmark results and use insights to tune and verify the model's accuracy.

Afterwards, the QoS extensions in the verified model were activated and it was used to develop configurations and interfacing approaches for virtualized hardware so that performance isolation for PCIe passthrough was achieved. This last step is subject to Sections 5.4 and 5.5. First, steps (1) to (4) are elaborated in detail in the following.

The whole model was realized with SystemC TLM 2.0, because it supports abstract modeling of busses and interconnects that are memory-mapped [132], which was a perfect match for the memory-mapped PCI Express interconnect. Additionally, TLM 2.0's generic payload and base protocol extension mechanism allowed easy modeling of the required protocol characteristics of PCIe. First, generic PCIe ports were modeled that support all QoS extensions that were presented in Section 5.2.2, together with configuration parameters for: (1) The number of ingress ports, (2) the number of VCs, and (3) the arbitration scheme selection for each arbiter of the Port and VC arbitration stages. If WRR or TBWRR arbitration schemes were selected for any of the arbitration stages, respective arbitration table data structures had to be supplied as well. Supplying one TC/VC mapping data structure per port was mandatory.

Afterwards, three generic, higher-level system components were modeled that each incorporated the PCIe port models. This way, all kinds of PCIe hierarchies could be modeled and assembled. The first component was a multi-core CPU with integrated Root Complex and a configurable number of cores. Second, a Platform Controller Hub (aka chipset) that acted as a packet relay, and third, an Endpoint with a configurable number of PCIe functions and Ethernet ports that could be assigned to each other. The Endpoint's PCIe functions could either resemble legacy functions or SR-IOV PFs and VFs. On the abstraction level of the SystemC model, there was no need to differentiate between function types, as they only differ in their ability to configure device characteristics and not in terms of performance.

The PCIe port models transmit and receive PCIe packets in accordance to the PCIe specification [78]. This was realized using the SystemC approximately-timed coding style and extending the SystemC generic payload with PCIe packet characteristics like the Traffic Class. Flow control mechanisms between two interconnected PCIe port models indicated free target buffers, so that transaction generation and sendout was blocked in case of congestion. Cores in the multi-core CPU were modeled as individual processes that directly insert into PCIe ports. This is an abstraction from the complex inner structure of modern x86 multi-core CPUs that was necessary for reducing complexity of the SystemC model and achieving reasonable simulation times.

In the next step, the models for the generic system components were used to create a model of a real-world lab-setup that utilizes SR-IOV Ethernet networking. To improve comparability, the lab-setup was almost identical to the one that was used in the previous chapter (compare Section 4.3.1). Instead of the six-core Xeon E5-2630**v1**, an E5-2630**v2** was utilized, which had a 300 MHz faster clock speed (2.6 GHz in total). Chipset (C602) and SR-IOV capable NIC (82576 dual-port Gigabit Ethernet) were identical. The lab-setup was configured to spawn two VMs on separate CPU cores, and three VFs on the SR-IOV NIC. VM0 was attached to VF0.0, the first VF of PF0. VM1 was either attached to VF0.1, which belonged to the same PF than VM0, or to VF1.0, which belonged to the other PF. Hence, depending on configuration, VM1 either shared an Ethernet port with VM0 (eth0), or not (eth1). This configuration is depicted in Figure 5.4.

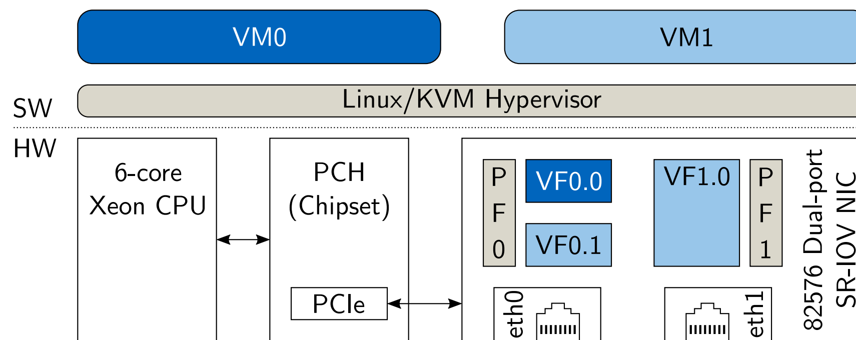


Figure 5.4: Block diagram of the real-world lab-setup running two VMs. VM0 was attached to VF0.0 of PF0. VM1 was either attached to VF0.1 (same PF as VM0) or VF1.0 (different PF).

The configuration of the lab-setup machine was recreated using the SystemC model. The model was also capable of computer-generating a graphical representation of the system components and their connections, which allowed convenient verification of the modeled hierarchy. The generated hierarchy is depicted in Figure 5.5.

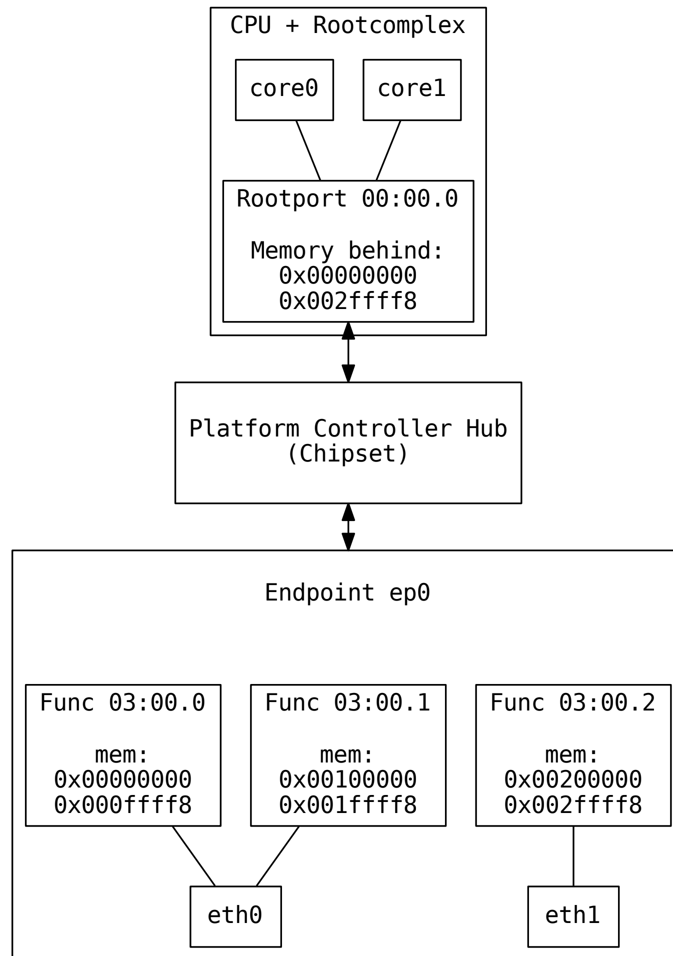


Figure 5.5: The PCIe hierarchy of the SystemC model, computer-generated by the code of the model itself at simulation runtime.

It shows core0 and core1 representing VM0 and VM1, as well as three PCIe functions in the Endpoint that are assigned to the respective Ethernet ports like the 82576 NIC VFs are in the lab-setup. The model is also able to automatically enumerate PCIe functions and to generate memory maps that are partitioned like in a real-world system. The generated address ranges can then be used by the CPU processes to send PCIe packets to specific functions.

5.3.1 Model Verification Approach

For the SystemC model to become a viable research platform for PCIe DoS attacks, it had to be able to accurately simulate the respective effects. Therefore, the goal was to simulate execution of DoS attacks and UDP streaming benchmarks on core0 and core1. On the model, UDP streaming was simulated by letting the model's cores and their attached PCIe functions communicate using the protocol that was described in Section 3.6. On the lab-setup, UDP streaming was benchmarked using netperf, just like in the previous chapters of this thesis. Measurements on the lab-setup were then used to annotate the timings for packet generation and processing delays in the SystemC model (using mean values from multiple runs).

Finally, UDP throughput results on both, lab-setup and SystemC model, were compared to verify the accuracy of the simulation. To avoid confusion, the following descriptions will only use names as depicted in the block diagram of the lab-setup (see Figure 5.4), but they can be substituted ad libitum with the respective names from the SystemC model hierarchy[3](compare Figure 5.5). The following three scenarios were evaluated:

Baseline: VM1 was completely idle and neither executing a benchmark nor a DoS attack. VM0 was the only user of Ethernet port eth0 and was executing the UDP throughput benchmark.

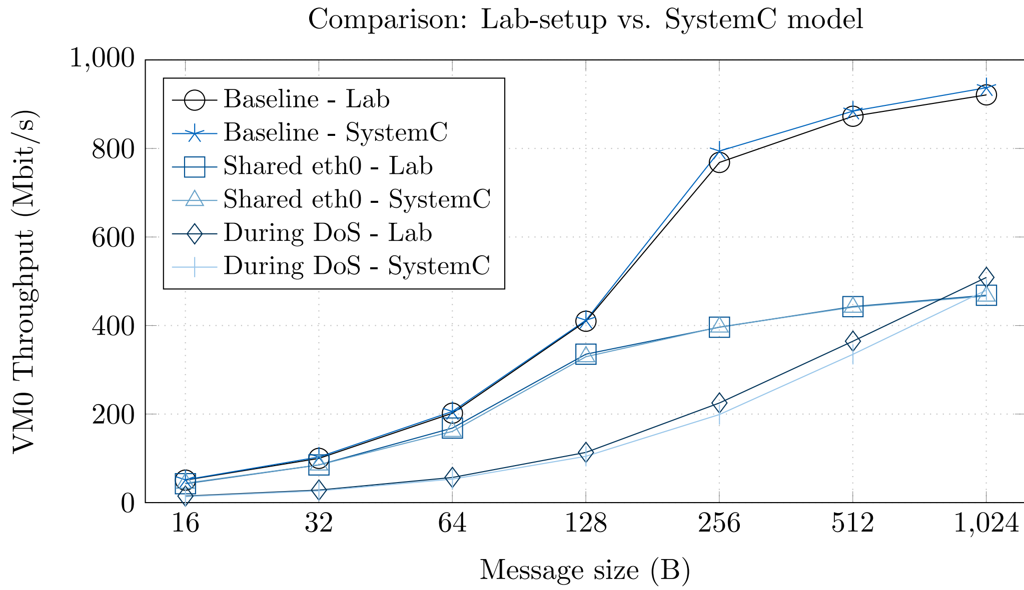
Shared eth0: VM1 was attached to the second PCIe function of eth0 (VF0.1), and was concurrently executing the same benchmark as VM0. Hence, eth0 was shared by both VM0 and VM1 simultaneously.

During DoS: VM1 was either attached to VF0.1 (same eth port than VM0) or VF1.0 (different eth port) and was executing a PCIe DoS attack. Both configurations yielded the same results and are therefore presented as a single result in the following. This behavior is an expected result for a COTS SR-IOV device that, most likely, uses the same input buffers (VC0 only in this case) for all its PCIe functions (PFs and VFs). It might therefore only have a single processing engine for all incoming MMIO write transactions, regardless of PCIe function type or number. This was modeled accordingly in the SystemC model.

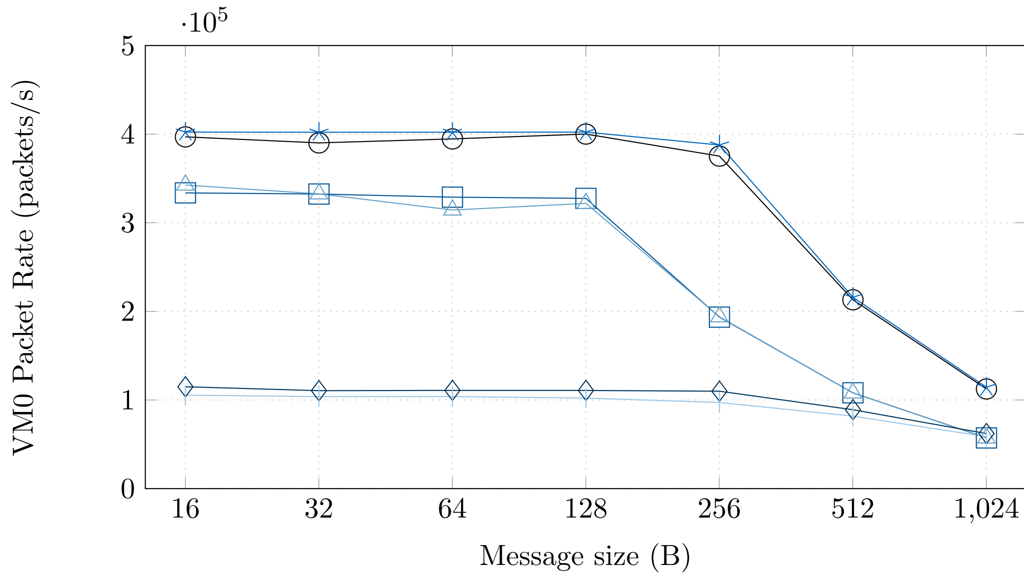
5.3.2 Verification Results

Figure 5.6 depicts results for UDP throughputs for the lab-setup and the SystemC model for a common range of message sizes (16 B – 1024 B). Additionally, the corresponding rate of generated UDP packets per second is depicted for both. This metric helps to further verify that the netperf benchmark was modeled accurately in the SystemC model.

[3]VM0 ↔ core0
VM1 ↔ core1
VF0.0 ↔ Func 03:00.0
VF0.1 ↔ Func 03:00.1
VF1.0 ↔ Func 03:00.2



(a) UDP throughput



(b) Packets per second

Figure 5.6: VM0's UDP throughput and corresponding packets per second on the lab-setup and the SystemC 2.0 TLM model while VM1 is either idle, executing the same benchmark as VM0 or executing a DoS attack.

Data points for results of the SystemC model each represent a one-time simulation run. Evaluation showed that a statistical distribution of single PCIe packet generation rates did not have a significant impact on the results. This was due to the communication protocol between CPU cores and I/O device, which involved a lot of blocking depending on the UDP message size. This resulted in the system converging into one of two possible states, in which either state produced blocking times that were much larger than variances in the PCIe packet generation of CPU cores. Both states are explained as follows:

I/O bound: UDP packets had large payload sizes. A CPU core needed less time to generate an UDP packet with a large payload than the I/O device needed to send out the respective packet. Hence, CPU cores were blocked and idle most of the time, waiting for the I/O device to finish sending the current packet before the next one could be generated.

CPU bound: UDP messages had small payload sizes. Creating large volumes of small packets incurred significant overhead for CPU cores. In consequence, a CPU core needed more time to generate an UDP packet with a small payload than the I/O device needed to send out the respective packet. Hence, the I/O device was blocked and idle most of the time, waiting for the CPU cores to deliver the next packet.

Results for the lab-setup show that DoS attacks caused UDP throughput drops between 45% (1024 B) and 72% (128 B). UDP throughput during a DoS attack was even worse than sharing the same Ethernet port (eth0 in this case) with another VM. According to the classification of DoS attacks that was presented in Section 3.7, the attack was therefore both an intra- and inter-PF DoS attack. Comparing these results to the respective results of the SystemC model shows that there is an average percentage error of 1.9% between both in the baseline case. It is 1% for the case where the Ethernet port is shared (VM0 and VM1 simultaneously utilized eth0), and 7.7% for the case where VM1 was executing a DoS attack. The errors can be attributed to the high-level abstractions in the model that were highlighted previously. Nonetheless, the SystemC model results were very close to those of the real-world lab-setup, so that the it provided a good basis for developing and exploring hardware architectures that employ PCIe's QoS extensions to prevent DoS attacks.

Two such hardware architectures will be presented in the following two sections. The first architecture is optimized for full scheduling freedom, the second for minimal hardware costs. Both architectures were thoroughly evaluated with help of the SystemC model, and the results will be discussed together with the pros and cons of each approach. In order to make evaluation results comparable, each architecture incorporated the SR-IOV two-port Gigabit Ethernet NIC that has been presented in previous sections of this thesis. However, the concepts are not limited to this special device or I/O category, but can be applied to any kind of passthrough hardware, e.g. a legacy PCIe device with multiple functions.

5.4 Multi-VC Architecture

The core idea of the first architecture is to resolve performance isolation issues by leveraging PCIe's option for multiple VCs. They are used to physically separate PCIe packet flows of malicious VMs from flows of legal, uncompromised VMs. As previous chapters in this thesis showed, it is the CPU-to-I/O-device direction, alternatively called the downstream direction, that is exploited by VMs for PCIe DoS attacks. However, there are also non-VM sources for downstream PCIe packets that had to be accounted for in the design of the architecture. These exploration results are presented in the following.

The block diagram in Figure 2.9 gives a good overview of a generic, virtualized system and its multiple sources of downstream PCIe packets. These are: (1) VMs, (2) the hypervisor or other trusted host software and (3) the DRAM controller of the CPU. VMs and hypervisor are software entities that execute on the same pieces of hardware, the cores of the CPU. In conclusion, there are only two types of hardware sources for PCIe downstream packets, the DRAM controller and the CPU cores. However, for the latter, it must be differentiated which kind of software is running, because it determines whether the generated PCIe packet might be part of a DoS attack flow or not. In other words, the software executing on a CPU core determines, at runtime, if a CPU core is a trusted (hypervisor executes on core) or an untrusted (VM executes on core) source of a PCIe packet. The DRAM controller, in turn, is always a trusted source. It does not run potentially malicious software, and only completes DMA requests of the I/O device. Since the I/O device knows its own bounds for processing PCIe packets, it would never issue more DMA requests than it can handle, so that a DoS attack caused by DMA requests can be ruled out. Hence, the DRAM controller can be considered a trusted source for downstream PCIe packets. Table 5.2 depicts these classifications for trusted and untrusted PCIe packet sources.

Table 5.2: Downstream PCIe packet sources classified by trust.

Hardware source	CPU core		DRAM
Software	Virtual Machine	hypervisor	N/A
Trusted?	✗	✓	✓

5.4.1 Architecture

The distinction of trusted and untrusted PCIe packet sources was used as a key aspect in the design of the multi-VC architecture. The performance isolation issues are resolved by physically separating the flow for trusted PCIe packets from each individual flow of untrusted packets using multiple Virtual Channels. Additionally, PCIe packets from trusted sources are prioritized over untrusted ones. With this architecture, backpressure from PCIe DoS attacks is only put on the attacker itself and not on concurrent VMs. A block diagram of the architecture is depicted in Figure 5.7.

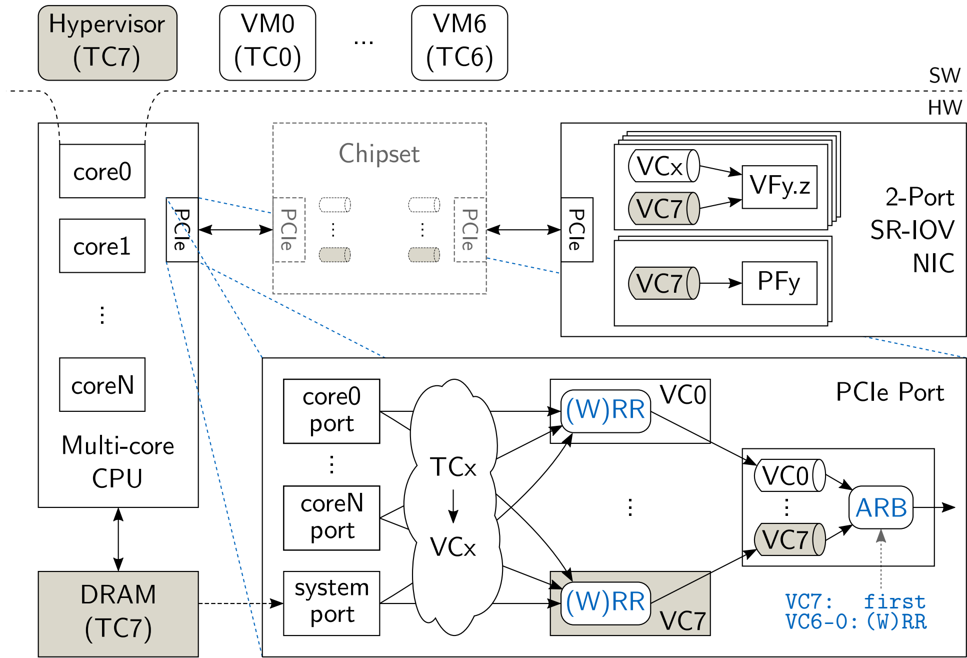


Figure 5.7: Multi-VC architecture: Trusted (hypervisor + DRAM controller) and untrusted (VMs) PCIe packet flows are separated with multiple VCs. Trusted sources get their own TC label, as well as each untrusted source (VMs). TC/VC mapping is done 1:1, enabling the separation of flows.

The separation of flows is realized with PCIe's Traffic Class (TC) labeling of PCIe transactions and a 1:1 TC/VC map. This means that transactions labeled with TC0 are assigned to VC0, TC1 to VC1, and so on. This mapping scheme ensures that each TC label is assigned its own and exclusively used VC buffers. In the architecture, the highest TC, TC7, is always assigned to transactions of the trusted sources, aka the DRAM controller and CPU cores that run the hypervisor software. Hence, they both share VC7, which is also configured to be the high-priority VC. The remaining low-priority VCs are assigned to the VMs. In the current revision of the PCIe standard, there are eight TCs and VCs. This means that using this partitioning, seven VCs remain for assignment to VMs (VC0–VC6).

In the PCIe port of the multi-core CPU, a two-stage arbitration scheme is employed. For Port Arbitration, a (W)RR scheme is used between the different input ports. For VC Arbitration, the high-priority VC7 (hypervisor + DRAM) is always selected first. If no transactions are pending for VC7, the remaining VCs are also selected in a (W)RR fashion. WRR instead of RR can be activated if weighted link partitioning between the low-priority VCs, aka VMs, is desired. In order for the architecture to function properly, there must be an identical number of VCs in the whole PCIe hierarchy. This means that at least the CPU PCIe port and the SR-IOV device must provide identical VCs. If an optional chipset is interposed between CPU and I/O device, it must also provide the respective number of VCs.

As previously mentioned in Section 5.2.1, the interface between the host CPU and the PCIe ports is not covered in the PCIe specification. For the proposed architecture, it is essential that the CPU's PCIe ports are able to distinguish between single cores. Only if this requirement is fulfilled, it is possible to put the back-pressure from a congested VC back on the attacker VM itself. This is because the port arbiter for each VC masks out internal ports whose pending arbitration requests target a VC that has no flow control credits left, e.g. because it is congested. This mechanism ensures that only cores are masked out on which a VM is running that executes a DoS attack, which eventually congests the respective VC. For this reason, each core is depicted as an internal port in the CPU's PCIe port diagram (see core0–coreN in Figure 5.7).

Additionally, the virtualized multi-core CPU must provide a mechanism for dynamically mapping TC labels to CPU cores, depending on which VM currently executes on a specific core. TC information could be incorporated into per-VM data structures that save processor state and configuration, and are evaluated each time a VM is scheduled on a new or different CPU core. For example, it would be possible to incorporate TC information within the Intel "Virtual Machine Control Data Structures" (VMCS) [4], which store register and non-register states of VMs. VMCS are part of the VT-x processor extensions [2], which are the hardware accelerators for virtualization. A hypervisor can maintain one VMCS per VM. AMD x86 processors provide a similar data structure, called "Virtual Machine Control Blocks" (VMCB) [133].

It is also important for the DRAM controller to have a distinct port and VC. Otherwise, it would need to share VCs with potentially malicious VMs. If a malicious VM executes a DoS attack, the VC might become congested, so that the DRAM controller might block for several arbitration cycles if it wants to insert a transaction. This time is missing for the next transaction of the DRAM controller, which might belong to a different VM, so that the performance of concurrent VMs might suffer. In Figure 5.7, this distinct port is called the "system port". It is not exclusive to the DRAM controller, but can be shared with any other system-level resource that is trusted and can be reached by the system's I/O devices via DMA.

Inside the I/O devices, there is no need for a special mapping of VCs to functions, since PCIe routing is based on memory addresses. Each PCIe function (legacy as well as PFs and VFs) is assigned a unique memory address range at system boot-up, so that an incoming PCIe transactions cannot have multiple targets. If the system has an optional chipset interposed between CPU and I/O device, the privileged VC7 can be shared with other trusted flows that target other (also non-virtualized) I/O devices that are connected to the chipset.

5.4.2 Evaluation

For evaluation of the proposed multi-VC architecture, it was modeled accordingly with the verified SystemC model. Afterwards, the same set of benchmark scenarios as in Section 5.3.1 was executed: VM0 was attached to a VF of the first Ethernet port (eth0) and executed UDP throughput tests. VM1, in turn, was either attached to a VF of the same Ethernet port than VM0, or the other, physically disjunct port (eth1), and

executed DoS attacks. Both VMs were assigned exclusively to their own VCs, VC0 and VC1, respectively. Results showed that the multi-VC architecture completely prevented any performance isolation breaches due to PCIe DoS attacks of VM1. In contrast to the architecture without multiple VCs, VM0 was able to achieve baseline performance during concurrent DoS attacks of VM1.

Performance degradation did not emerge because the multiple VCs prevented that the attacker VM1 could insert a significant number of spurious PCIe packets ahead of each PCIe packet from VM0 (compare Figure 3.11). Instead, only VC1 was congested with DoS packets, while VC0 stayed free of them. This way, the processing facilities of the SR-IOV device were able to serve the VC buffers of the VMs and the DRAM controller in a fair Round-Robin fashion and not in a first-come first-served manner. In conclusion, at most one spurious DoS packet from the VC1 buffers got served by the I/O device before each legal packet from the VC0 or VC7 (DRAM controller) buffers. Because the delay of processing a single VM1 packet did not suffice to degrade performance of VM0, the DoS attack was completely prevented.

Given this design of an SR-IOV device, it would still be possible for an attacker to influence the processing time of packets of concurrent VMs. If an attacker controls enough VMs on the system, and therefore VCs, and floods all of them with PCIe DoS attacks, the processing facilities of the SR-IOV device could still be delayed long enough so that a certain VM's PCIe packets cannot be served in time (because all the flooded VCs must be served beforehand). Evaluations with the SystemC model found the critical number of VCs for the given SR-IOV device to be five or greater. Results for this test are depicted in Figure 5.8.

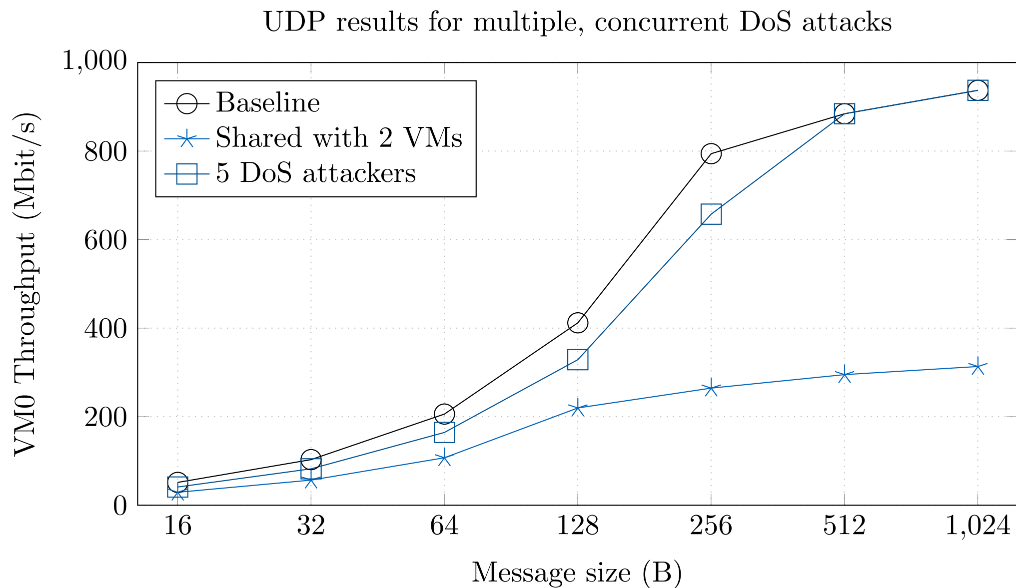
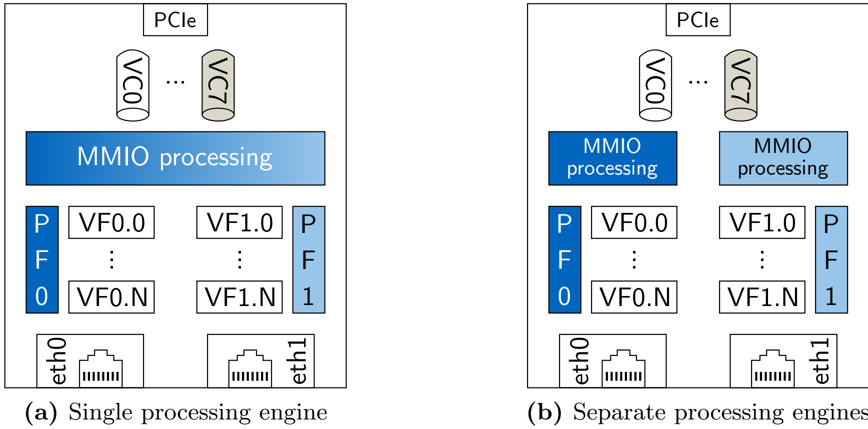


Figure 5.8: If the SR-IOV device employs the same packet processing facilities for functions of both Ethernet ports (eth0 and eth1), then it is possible to cause an inter-PF attack with a simultaneous DoS attack of five or more attacker VMs.

The results also show that sharing the same Ethernet port with two legal VMs took away more UDP throughput than the five simultaneous DoS attacks. This attack is, according to the DoS attack classification introduced in Section 3.7, therefore, only viable as an inter-PF attack. One such scenario would be, for example, if DoS attacks on five VFs of the first Ethernet port (eth0, PF0) cause performance degradation of a VF of the second Ethernet port (eth1, PF1). Such an inter-PF attack is only possible if the SR-IOV device employs the same processing facilities for incoming MMIO PCIe packets for both PFs and their corresponding VFs. This attack vector can be closed by design. Therefore, each PF and its VFs must be assigned to physically separate processing facilities. A comparison of both design approaches is depicted in Figure 5.9.



(a) Single processing engine

(b) Separate processing engines

Figure 5.9: Comparison of an SR-IOV device design with a single processing engine for all PFs/VFs (a) and separate processing engines per PF (b).

Employing the second approach (b), flows for different PFs are completely decoupled. Using separate processing engines for both PFs, it is not possible to influence any of PF0's VFs by flooding VFs from PF1, and vice versa. Modeling the SR-IOV device according to design (b) in the SystemC model confirmed that this design approach closed the observed attack vector for inter-PF attacks.

5.4.3 Multi-VC Architecture Summary

The proposed multi-VC architecture utilizes multiple VCs to separate PCIe flows of different VMs, preventing performance degradation due to DoS attacks. Flows are differentiated by dynamically tagging VMs with unique TC labels and a 1:1 TC/VC mapping. The approach preserves full scheduling freedom: VMs can be scheduled on an arbitrary number of cores at any time. In terms of hardware costs, the architecture may become costly. The whole PCIe hierarchy (CPU, optional chipset and I/O device) must employ the same amount of hardware VC buffers. For full hardening, separate processing engines per PF are needed inside SR-IOV devices. Scalability is constrained by the maximum number of VCs that is specified in the PCIe specification (eight at the moment [78]).

5.5 Time-based Weighted Round-Robin Architecture

The second architecture proposed in this chapter is called the Time-based Weighted Round-Robin (TBWRR) architecture. It was developed as an alternative to the previously introduced multi-VC architecture. In contrast to the latter, less resources are needed for the TBWRR architecture, making it more light-weight. Additionally, only the virtualized multi-core CPU needs changes to hardware and the architecture utilizes only a single VC, which makes it compatible to legacy chipsets and passthrough I/O devices. The architecture leverages a TBWRR arbitration table for Port Arbitration that is tailored to the respective passthrough I/O device in order to prevent performance degradation due to PCIe DoS attacks. For the arbitration scheme to work properly, some constraints must be put on VM scheduling. The core idea of the architecture will be elaborated in the following.

Recalling Figure 3.11, performance degradation due to PCIe DoS attacks emerges because the I/O device is overstrained: The source (CPU cores) is sending PCIe packets at a rate that is higher than the processing rate of the packets at the sink (I/O device). This eventually leads to a congestion on the interconnect that degrades performance for concurrent VMs and the host. In current virtualized passthrough systems, there is no possibility to constrain the sending rate at the source. If the source rate could be constrained such that $R_{Source} \leq R_{Sink}$, then the DoS attack vector would be eliminated. R_{Sink} for real-world hardware can be found either by looking at the respective device's datasheet or experimentally measured by trusted host software (see Appendix A) before the VMs are instantiated. For example, the I/O device that was used in this chapter, the 82576 dual-port SR-IOV capable Gigabit Ethernet NIC, was targeted by DoS attacks at a VF memory address that took **534 ns** to process a single packet. To leverage this information, a virtualized multi-core CPU architecture is needed that is able to constrain its cores so that their packet rate is slower than 534 ns per packet.

5.5.1 Architecture

This can be achieved with the TBWRR Arbitration scheme that is defined in the QoS extensions of the PCIe standard [78]. It is the only arbitration scheme that enables fine-grained temporal control of single PCIe transactions. TBWRR is only available for the Port Arbitration stage (see Figure 5.2). However, this is an advantage, because it enables the whole architecture to work with only a single VC, VC0, which makes the architecture compatible to legacy chipsets and I/O devices. A block diagram of the proposed architecture is depicted in Figure 5.10.

The considerations about trusted and untrusted PCIe packet sources from the previous section were reused for the design of this architecture. Internal ports for CPU cores were considered untrusted, because they issue PCIe packets on behalf of potentially malicious VMs. This means that the time between the arbitration of CPU cores must be greater than the 534 ns that are needed for the SR-IOV device to process a PCIe packet. According to the standard, the duration of a TBWRR timeslot is 100 ns [78]. Hence, every sixth timeslot in the TBWRR arbitration table may be used by a CPU

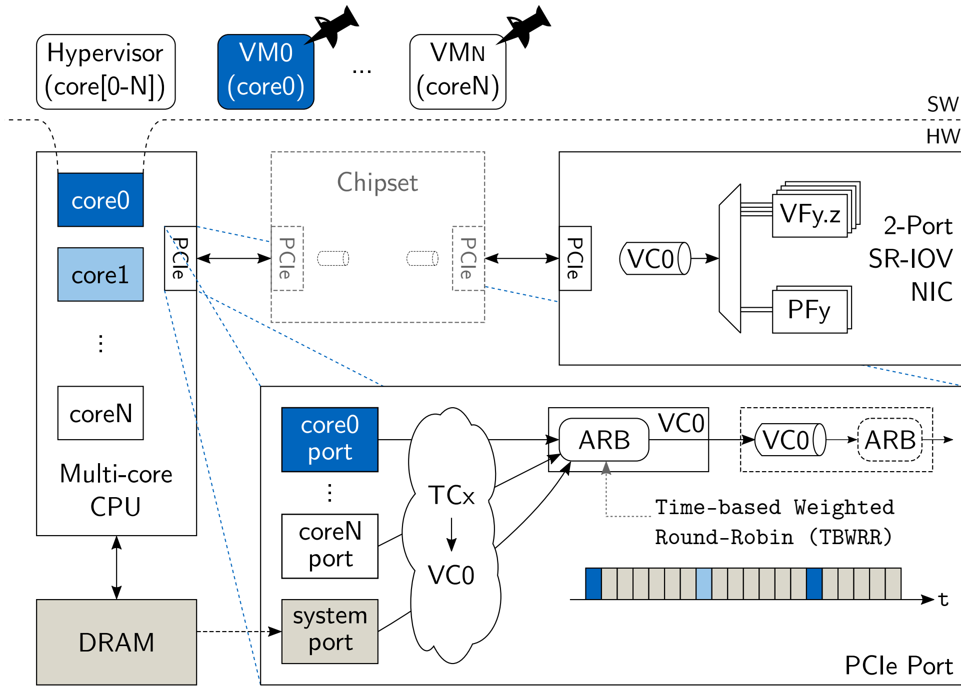


Figure 5.10: The TBWRR architecture. DoS attacks are eliminated by a Port Arbitration table that prevents CPU cores from sending PCIe packets at rates that are higher than the packet processing rate of the SR-IOV device.

core. As TBWRR arbitration works with internal ports, it must differentiate the cores of a multi-core CPU using one port per core. Fair allocation of resources can therefore only be guaranteed if VMs are statically pinned to certain cores. The remaining five arbitration timeslots between two CPU core slots are used for transactions of other system components. For example, for transactions of the DRAM controller that complete DMA requests from I/O devices. The hypervisor does not need special treatment, because it executes on any core that also executes VMs. It can therefore utilize the arbitration slots of the respective VM. An exemplary TBWRR arbitration table is shown in Figure 5.11.

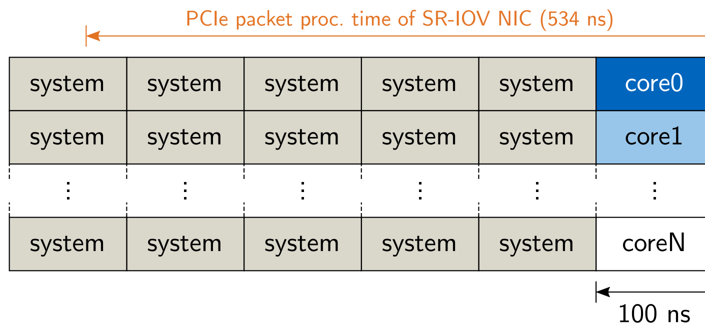


Figure 5.11: TBWRR arbitration table, tailored for preventing DoS attacks on the 82576 NIC.

5.5.2 Evaluation

The TBWRR architecture was modeled with help of the SystemC model and evaluated with the same set of benchmark runs as the multi-VC architecture in the previous section. Results showed that the TBWRR arbitration was able to deliver full performance isolation during PCIe DoS attacks. It was not possible for malicious attacker VMs to congest the interconnect, because the SR-IOV device could not be overstrained thanks to the throttling of the arbitration table. The attack vector for flooding VFs was completely closed. Unfortunately, there is a small performance price to pay, because the timeslot quantization of the arbitration table has a small impact on the maximum achievable performance of individual VMs. Sending back-to-back packets with an TBWRR scheme is not as fast as a (legacy) best effort scheme, because there can be only one transaction every 100 ns. The throughput difference between TBWRR and best effort arbitration for multiple VMs sharing a PF is depicted in Figure 5.12.

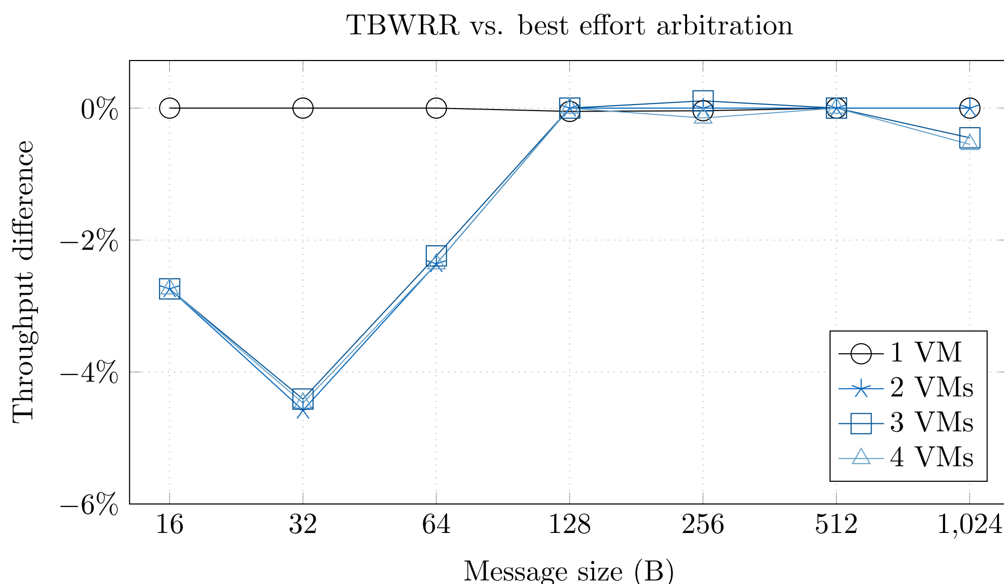


Figure 5.12: The difference in UDP throughput of the TBWRR arbitration scheme compared to best effort. Legend entries indicate the number of VMs that shared a Physical Function of the simulated 82576 NIC. Similar results were also achieved for more than four VMs.

For small message sizes ($\leq$ 64 B) and at least two VMs that shared a PF, there was a small degradation in overall performance for TBWRR compared to best effort. Bigger message sizes saw almost no impact. In conclusion, the TBWRR architecture is able to successfully provide performance isolation against PCIe DoS attacks, while delivering between 100% and 95.4% of best effort arbitration performance.

The throughput results in Figure 5.12 covered the case that all VMs were constantly trying to send on the Ethernet port that they share with their concurrent VMs. However, it may also be possible that some VMs are temporarily idle and therefore do not utilize

their shared resource. Unfortunately, this leads to scenarios where it is not possible for other VMs to utilize these spare resources due to the static nature of the TBWRR table that is depicted in Figure 5.11. In order to achieve peak performance, it is necessary that an Ethernet port receives new descriptor updates from VMs at certain minimum rates. Otherwise, it cannot fetch new Ethernet packet contents from DRAM fast enough to saturate the Ethernet link. This descriptor update rate depends on the message size of the Ethernet packets. Big message sizes need fewer Ethernet packets for sending the same payload than small message sizes. For the lab-setup (and the SystemC model) that was used in this chapter, the peak for descriptor updates was reached at message sizes being equal or smaller than 128 B. The rate was measured at round about 400000 descriptors per second, or one descriptor each 2.5 µs.

Figure 5.13 shows an extended version of the arbitration table of Figure 5.11. It depicts a scenario in which only one CPU core is active and other cores are idle, and has additional scheduling times indicated on the right. If only core0 is active and sending out new descriptor updates, while cores1-4 are idle, then it is not possible for the VM running on core0 to achieve the descriptor update rate of 2.5 µs that is needed for maximum utilization of the lab-setup's NIC. In the depicted scenario, a new descriptor would be generated only once every 3 µs.

system	system	system	system	system	core0 (active)	0 ns
system	system	system	system	system	core1 (idle)	600 ns
system	system	system	system	system	core2 (idle)	1200 ns
system	system	system	system	system	core3 (idle)	1800 ns
system	system	system	system	system	core4 (idle)	2400 ns
					100 ns	

Figure 5.13: TBWRR arbitration table with idle cores. Cores1-4 do not utilize their slot for sending because they are idle, which results in core0 sending one PCIe packet (new descriptor update) to the 82576 NIC every 3 µs.

If this case is of concern, it can be resolved with an extension to the TBWRR architecture (Figure 5.10) that is transparent to the Port Arbitration stage and therefore compatible to the QoS extensions of the PCIe specification. Instead of statically allocating CPU cores in the TBWRR table without knowledge about pending transactions, a two-stage arbitration can be implemented. If a (weighted) Round-Robin arbiter for CPU cores precedes the TBWRR arbitration stage, cores that have pending transactions can be pre-selected. The TBWRR arbitration table, in turn, is reduced from multiple rows containing explicit CPU core slots to only a single row containing a generic CPU core slot. A pending transaction of the CPU core arbiter can then be forwarded once the Port Arbiter selects the generic CPU core slot in the TBWRR table. This mechanism prevents that idle cores block a CPU slot and guarantees full utilization of spare resources. The concept is depicted in Figure 5.14.

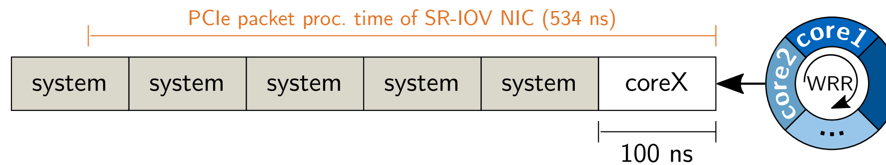


Figure 5.14: TBWRR arbitration table with pre-selection of CPU cores. This approach skips idle cores in favor of cores with pending PCIe transactions.

5.5.3 TBWRR Architecture Summary

The proposed TBWRR architecture was designed as a light-weight alternative to the multi-VC architecture. It requires a virtualized multi-core CPU that supports the TBWRR arbitration scheme for Port Arbitration. If full utilization of spare resources is desired, a (weighted) Round-Robin scheduling stage for CPU cores with pending PCIe transactions is needed as well. The architecture does not need changes to other hardware than the CPU, making it compatible to existing legacy chipsets and I/O devices. Additional VC buffers are not needed. The timeslot quantization of the TBWRR arbitration table may introduce small performance overheads, however, they depend on the protocol that is employed between I/O device and CPU cores. The TBWRR table must be tailored to the specific I/O devices that are used by the VMs. This makes the approach better suited for point-to-point connections between CPU and I/O device, rather than sharing PCIe links with multiple devices, e.g. via an interposed chipset.

For fair sharing of the resources, it is mandatory that VMs are pinned to specific cores. Scalability of the architecture may be constrained by the maximum number of slots that is supported by the TBWRR table, which is 256 in the current revision of the PCIe standard [78].

5.6 Summary

PCIe passthrough and SR-IOV are suitable technologies for enabling hardware-assisted, low-latency I/O virtualization in future embedded and mixed-criticality multi-core systems. Unfortunately, in current implementations, there is a vulnerability to PCIe DoS attacks of malfunctioning or malicious VMs. DoS attacks can significantly increase latencies (see Chapter 3) and therefore degrade the performance of other, concurrent VMs. In the previous chapter, a reactive mitigation solution was presented for the cloud computing domain. For embedded and mixed-criticality systems, though, a proactive solution that completely closes the DoS attack vector is preferred, because remaining performance fluctuations during DoS mitigation must not be accounted for in WCET calculations.

This chapter presented research on hardware architectures that prevent DoS attacks and enable full performance isolation for PCIe passthrough and SR-IOV. Initially, domain-specific goals and requirements were collected. Performance isolation had to be achieved by extending existing hardware architectures while staying fully compatible to current PCIe and SR-IOV specifications at the same time. The hardware architectures had

to satisfy requirements for proactive, full isolation, secure configuration by a trusted software entity like the hypervisor, scalability to a reasonable number of passthrough functions and low overhead regarding non-PCIe parts of the system.

The developed architectures were built around optional QoS extensions of the PCIe specification, which facilitated compatibility. During exploration and review of the QoS extensions, it was found that certain hardware building blocks (Virtual Channel buffers, Port and VC arbiters and corresponding arbitration schemes) could be directly employed. On the other hand, paradigms for using these building blocks for prevention of downstream congestion (due to DoS attacks) were missing as well as mechanisms for interfacing the QoS hardware from a virtualized multi-core CPU. Both aspects were developed from scratch in this chapter.

Since current COTS PCIe hardware did not provide any QoS extensions (defined optional in the specification and lack of use cases in the real world), a SystemC TLM 2.0 simulation model of a real-world lab-setup was developed and extended with configurable PCIe QoS extensions. The SystemC model's accuracy was verified by replicating various UDP throughput benchmarks on it that were also run on the real-world machine. Comparison of the results showed that the average percentage error between software model and real-world hardware was between 1% and 7.7%.

Afterwards, the SystemC model was used to design and evaluate two hardware architectures that close the PCIe DoS attack vector. Evaluations showed that both architectures succeeded in completely preventing PCIe DoS attacks, while being optimized for different goals. The multi-VC architecture provides full scheduling freedom, but needs to implement multiple Virtual Channels in all PCIe devices of the hierarchy (multi-core CPU, optional chipset, I/O device). The lightweight TBWRR architecture, on the other hand, only needs hardware changes to the multi-core CPU, but VMs must be pinned to specific cores. Detailed reviews of both architectures can be found in Sections 5.4.3 and 5.5.3, respectively.

6 Conclusion and Outlook

Work presented in this thesis contributes to the field of performance isolation for PCIe passthrough and Single Root I/O Virtualization (SR-IOV), the latest generation of I/O virtualization technology. Specifically, performance isolation issues that arise from PCIe Denial-of-Service (DoS) attacks on passthrough and SR-IOV hardware were addressed. The focus was laid on the cloud computing domain, where the technology has been embraced and deployed already, and the embedded virtualization domain, where the technology is seen as an enabler for future, virtualized multi-core systems.

6.1 Conclusion

Initial work on this thesis was motivated by the sparse presence of literature on performance isolation of PCIe passthrough and SR-IOV. It quickly led to the discovery that PCIe DoS attacks of malicious or buggy Virtual Machines (VMs) are able to break performance isolation in current commercial-off-the-shelf (COTS) hardware that implements the technology. This attack vector was investigated with four experiments on a state-of-the-art x86 system comprised of COTS hardware, including a dual-port 1 Gbit/s SR-IOV capable Ethernet adapter. Results showed that DoS attacker VMs were able to significantly degrade the I/O performance of concurrent VMs in configurations where the victims should have been 100% isolated from the attacker. For example, PCIe DoS attacks on the Ethernet adapter increased latencies for accessing the device from 1.58 µs up to 12.61 µs (+698%). This translated to an Ethernet throughput drop from 941 Mbit/s down to 615 Mbit/s (-35%). The same attacks caused the system's disk storage throughput to drop by up to 77%.

Learnings from the experiments were used to identify the key architectural issues within the design of PCIe passthrough and SR-IOV: First, CPU cores, on which potentially malicious VMs execute, are usually faster at generating PCIe packets than passthrough I/O devices are at consuming them. Second, in current implementations, there is no mechanism for passthrough devices to prevent VMs from sending packets. The resulting congestion on the shared PCIe interconnect causes performance interference between otherwise unrelated VMs. The gained insights were used to construct an abstract system model that presents a concise representation of these issues. Furthermore, four different DoS attack types were identified and classified according to the system components that suffer from the attack. For instance, it is differentiated if degradation of an attack affects only the shared device that is attacked, or also unrelated devices on the same interconnect.

In order to solve the performance isolation issues, two domain specific solutions were proposed. **The first solution targets the cloud computing domain** and proposes a

combined hardware/software approach. It was experimentally verified that lightweight hardware monitoring facilities within the passthrough I/O device are able to unambiguously detect DoS attacks and identify attacker VMs using a threshold value for transaction rates of PCIe packets. Additionally, two countermeasures that mitigate an attack were presented and evaluated. First, freezing attacker VMs showed that baseline performance for victim VMs is instantly restored, but the approach might also kill other, unrelated processes within the attacker VM. Second, attacker VMs can be throttled by scheduling them only for small timeslices that do not suffice to maintain a constant congestion on the interconnect. The latter approach keeps the VM alive and is able to restore close to baseline performance for victim VMs. For instance, performance of typical cloud applications like Apache and Memcached could be restored from 51.3% during DoS attacks to 97% and from 61.8% to 99.4%, respectively; Streaming-based micro-benchmarks performed only slightly worse.

The mitigation approach does not reach 100% of native performance for two reasons. (1) It is a reactive approach that can be invoked only after an attack has been detected, and (2) throttling via software scheduling could not reach granularities that were fine-grained enough to completely prevent congestion on the interconnect. However, this design decision was made on purpose in light of cloud computing requirements. Here, real-time capabilities are not needed, so that small interferences are tolerable. Even more so because cloud computing providers can migrate attacker VMs to isolated hosts shortly after a DoS attack was detected and mitigated. On the other hand, using a monitoring approach enables the proposed solution to be scalable and low-cost, which increases the chance of manufacturers adopting the idea.

The second solution targets the embedded virtualization domain and proposes two integrated hardware architectures that completely prevent DoS attacks on PCIe passthrough and SR-IOV. This way, performance interference of passthrough devices need not be considered at all in worst case execution time calculations of real-time systems. In order to stay compatible to PCIe, both architectures were designed to employ optional Quality-of-Service (QoS) extensions from the PCIe specification as a foundation. The QoS extensions originate from a time long before PCIe passthrough, SR-IOV and platform virtualization surfaced in datacenters and COTS devices, and DoS attacks on PCIe devices became a valid attack vector. Hence, this thesis shows a way to repurpose the QoS extensions in order to prevent DoS attacks.

The first proposed architecture, called multi-VC, maps individual VMs to dedicated Virtual Channels (VCs), which are implemented in CPUs, switches and I/O devices. This separation approach ensures that malicious VMs cannot congest the shared interconnect. Evaluation with a SystemC model verified full prevention of DoS attacks. The PCIe interconnect stays congestion free, and performance remains unaffected. However, hardware VCs might be expensive, and scalability is capped by the number of VCs that the PCIe specification supports (eight VCs). The second proposed architecture, called TBWRR, employs hardware-enforced time-based weighted round robin (TBWRR) arbitration for admission of VMs to the interconnect. The thesis shows that arbitration rules can be compiled that completely prevent congestion on the interconnect if VMs are statically pinned to CPU cores. Evaluation with SystemC showed that the approach

achieves I/O performance between 95.4% and 100% of best effort arbitration. The TB-WRR architecture is more lightweight and scalable than the multi-VC architecture, and compatible to legacy switches and I/O devices. Hardware changes need only be made to the multi-core CPU.

A comparison of the characteristics of the mitigation and isolation solutions that were presented in this thesis is depicted in Figures 6.1 and 6.2.

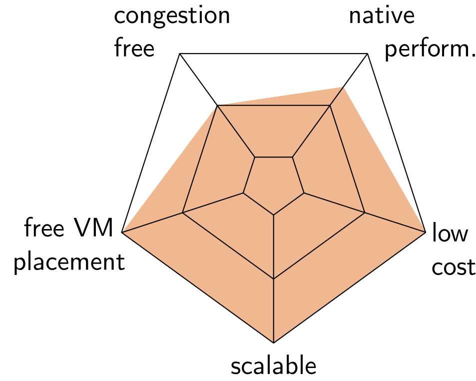
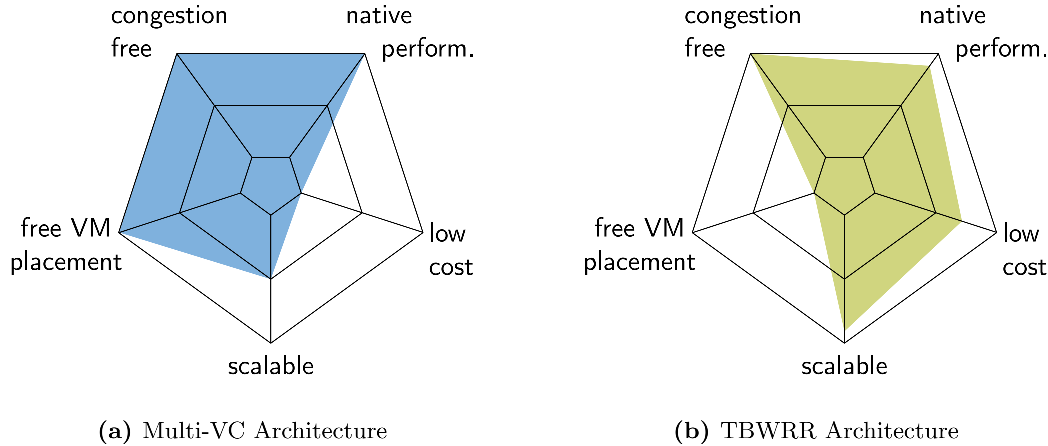


Figure 6.1: Characteristics of the cloud computing solution.



(a) Multi-VC Architecture

(b) TBWRR Architecture

Figure 6.2: Characteristics of the solutions for embedded virtualization.

The diagram for the cloud computing solutions reflects the previously argued design choices. Performance interference due to DoS attacks can be sufficiently mitigated until attacker VMs are taken care of by the provider. Pinning of VMs to cores is not needed, scalability is not constrained and implementation costs for monitoring facilities inside the I/O device are low. In contrast, the proposed architectures for the embedded domain focus on 100% freedom from interconnect congestion and therefore performance interference. The difference between the multi-VC and the TBWRR architecture is most visible

by the trade-off between implementation costs and freedom of VM placement. The multi-VC architecture is more expensive in terms of hardware implementation costs, but does not impose pinning of VMs to specific CPU cores. The TBWRR architecture is more affordable, but requires pinning of VMs by the hypervisor.

This thesis contributes a first step towards understanding the causes and implications of DoS attacks on PCIe passthrough and SR-IOV, the third generation of I/O virtualization technology. Additionally, first approaches for tackling the resulting performance isolation issues were presented by means of two domain-specific solutions. In future work, more research can be done to both deepen the understanding of performance isolation issues of PCIe passthrough and SR-IOV, and to improve the isolation solutions.

6.2 Outlook

There are many possible directions for contributing future work on the topic. For instance, different hardware could be employed to further evaluate the impact of DoS attacks on PCIe passthrough and SR-IOV, e.g. different I/O devices. Possibilities include faster Ethernet adapters or different classes of I/O, like Infiniband or SR-IOV capable graphics processing units, once they are released. In general, I/O devices that feature a higher PCIe bandwidth than the 1 Gbit/s Ethernet adapter that was primarily used in this thesis could be worth investigating. Preliminary tests of DoS attacks on a 10 Gbit/s SR-IOV capable Ethernet adapter on the lab-setups of this thesis hinted on performance interference effects that were more severe than for the 1 Gbit/s version, and additionally affected the main memory (RAM) performance of the host system. This could be a path worth pursuing. Research could also cover other processor architectures for the host platforms, e.g. ARM-based host systems.

Additionally, research on further improving the presented isolation solutions can be done. For example, various combinations of the three presented approaches are possible to overcome their individual weaknesses. In a mixed-criticality setup, a mix of mitigation scheduling and multiple VCs could be worthwhile. High-priority or real-time VMs could be mapped on dedicated VCs, while best-effort VMs share a single VC that is protected by the mitigation scheduling approach. The resulting design could provide a good balance between costs and isolation capabilities.

The time-based weighted round-robin (TBWRR) architecture for embedded systems could be improved by finding ways to arbitrate on VM IDs rather than core-specific ports. This would eliminate the current requirement of pinning VMs to specific CPU cores and introduce additional scheduling freedom. Here, the challenge is to maintain compatibility to the PCIe specification.

In the solution for the cloud computing domain, software scheduling is used to prevent congestion on the interconnect, which mitigates the DoS attacks. This part could also be offloaded to hardware-assist in the virtualized I/O device. If the I/O device detects a stream of malicious PCIe packets, a fast-path for packet processing could be invoked that discards PCIe packets of the malicious stream faster than an attacker VM can produce new ones, so that congestion cannot emerge.

Direct access to I/O devices or peripherals for VMs is also possible in virtualized platforms that do not employ PCIe as interconnect. Embedded systems often employ a multitude of different interconnects. Here, an evaluation of DoS attacks could contribute valuable insights, as well as research on transferring concepts of the presented solutions.

Bibliography

[1] L. Wang, G. Von Laszewski, A. Younge, X. He, M. Kunze, J. Tao, and C. Fu. Cloud computing: a perspective study. *New Generation Computing*, 28(2):137–146, 2010.

[2] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[3] Advanced Micro Devices, Inc. *AMD64 Virtualization Technology Secure Virtual Machine Architecture Reference Manual*, 3.02 edition, December 2005.

[4] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, December 2016. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D.

[5] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 26–35, New York, NY, USA, 2008. ACM. doi:10.1145/1346281.1346286.

[6] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*, pages 200–209, April 2007. doi:10.1109/ISPASS.2007.363750.

[7] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the Performance Isolation Properties of Virtualization Systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM. doi:10.1145/1281700.1281706.

[8] G. Heiser. Virtualizing embedded systems - why bother? In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 901–905, June 2011.

[9] D. Reinhardt, D. Kaule, and M. Kucera. Achieving a scalable e/e-architecture using autosar and virtualization. *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.*, 6:489–497, 04 2013. doi:10.4271/2013-01-1399.

[10] D. Reinhardt and G. Morgan. An embedded hypervisor for safety-relevant automotive e/e-systems. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 189–198, June 2014. doi:10.1109/SIES.2014.6871203.

[11] D. Juergens, D. Reinhardt, R. Schneider, G. Hofstetter, U. Dannebaum, and A. Graf. Implementing mixed criticality software integration on multicore - a cost model and the lessons learned. In *SAE Technical Paper*. SAE International, 04 2015. doi:10.4271/2015-01-0266.

[12] O. Sander, T. Sandmann, V. V. Duy, S. Bähr, F. Bapp, J. Becker, H. U. Michel, D. Kaule, D. Adam, E. Lübbers, J. Hairbucher, A. Richter, C. Herber, and A. Herkersdorf. Hardware virtualization support for shared resources in mixed-criticality multicore systems. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Dresden, Germany, March 2014. doi:10.7873/DATE.2014.081.

[13] D. Kleidermacher and M. Wolf. Mils virtualization for integrated modular avionics. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1.C.3–1–1.C.3–8, Oct 2008. doi:10.1109/DASC.2008.4702759.

[14] S. H. VanderLeest. Arinc 653 hypervisor. In *29th Digital Avionics Systems Conference*, pages 5.E.2–1–5.E.2–20, Oct 2010. doi:10.1109/DASC.2010.5655298.

[15] S. Han and H. W. Jin. Full virtualization based arinc 653 partitioning. In *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, pages 7E1–1–7E1–11, Oct 2011. doi:10.1109/DASC.2011.6096132.

[16] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling. Multicore in real-time systems–temporal isolation challenges due to shared resources. In *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems*, 2014.

[17] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache qos: From concept to reality in the intel xeon processor e5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 657–668, March 2016. doi:10.1109/HPCA.2016.7446102.

[18] J. Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 gbe nics with sr-iov support. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010. doi:10.1109/IPDPS.2010.5470365.

[19] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. *Journal of Parallel and Distributed Computing*, 72(11):1471 – 1480, 2012. Communication Architectures for Scalable Systems. doi:10.1016/j.jpdc.2012.01.020.

[20] S. Huang and I. Baldine. Performance evaluation of 10ge nics with sr-iov support: I/o virtualization and network stack optimizations. In *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance: 16th International GI/ITG Conference*, pages 197–205. Springer Berlin Heidelberg, march 2012. `doi:10.1007/978-3-642-28540-0_14`.

[21] G. Somani and S. Chaudhary. Application performance isolation in virtualization. In *2009 IEEE International Conference on Cloud Computing*, pages 41–48, Sept 2009. `doi:10.1109/CLOUD.2009.78`.

[22] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 51–58, July 2010. `doi:10.1109/CLOUD.2010.65`.

[23] G. Casale, S. Kraft, and D. Krishnamurthy. A model of storage i/o performance interference in virtualized systems. In *2011 31st International Conference on Distributed Computing Systems Workshops*, pages 34–39, June 2011. `doi:10.1109/ICDCSW.2011.46`.

[24] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao. Who is your neighbor: Net i/o performance interference in virtualized clouds. *IEEE Transactions on Services Computing*, 6(3):314–329, July 2013. `doi:10.1109/TSC.2012.2`.

[25] D. Gupta, R. Gardner, and L. Cherkasova. Xenmon: Qos monitoring and performance profiling tool. Technical report, Hewlett-Packard Labs, 2005.

[26] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 342–362, New York, NY, USA, 2006. Springer-Verlag New York, Inc.

[27] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: Communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 126–136, New York, NY, USA, 2007. ACM. `doi:10.1145/1254810.1254828`.

[28] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for i/o performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 101–110, New York, NY, USA, 2009. ACM. `doi:10.1145/1508293.1508308`.

[29] A. Gulati, A. Merchant, and P. J. Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 437–450, Berkeley, CA, USA, 2010. USENIX Association.

[30] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 1–10, New York, NY, USA, 2008. ACM. doi:10.1145/1346256.1346258.

[31] Y. Mei, L. Liu, X. Pu, and S. Sivathanu. Performance measurements and analysis of network i/o applications in virtualized cloud. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 59–66, July 2010. doi:10.1109/CLOUD.2010.74.

[32] I. S. Moreno, R. Yang, J. Xu, and T. Wo. Improved energy-efficiency in cloud datacenters with interference-aware virtual machine placement. In *Autonomous Decentralized Systems (ISADS), 2013 IEEE Eleventh International Symposium on*, pages 1–8. IEEE, 2013.

[33] Z. Yang, H. Fang, Y. Wu, C. Li, B. Zhao, and H. H. Huang. Understanding the effects of hypervisor i/o scheduling for virtual machine performance interference. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 34–41. IEEE, 2012.

[34] R. C. Chiang, S. Rajasekaran, N. Zhang, and H. H. Huang. Swiper: Exploiting virtual machine vulnerability in third-party clouds with competition for i/o resources. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1732–1742, June 2015. doi:10.1109/TPDS.2014.2325564.

[35] A. Richter, C. Herber, H. Rauchfuss, T. Wild, and A. Herkersdorf. Performance isolation exposure in virtualized platforms with pci passthrough i/o sharing. In *ARCS 2014 – International Conference on Architecture of Computing Systems*, pages 171–182, Lübeck, Germany, February 2014. Springer International Publishing. doi:10.1007/978-3-319-04891-8_15.

[36] A. Richter, C. Herber, T. Wild, and A. Herkersdorf. Denial-of-service attacks on pci passthrough devices: Demonstrating the impact on network- and storage-i/o performance. *Journal of Systems Architecture*, 61(10):592 – 599, November 2015. doi:http://dx.doi.org/10.1016/j.sysarc.2015.07.003.

[37] A. Richter, C. Herber, S. Wallentowitz, T. Wild, and A. Herkersdorf. A hardware/software approach for mitigating performance interference effects in virtualized environments using sr-iov. In *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, pages 950–957, New York, USA, June 2015. doi:10.1109/CLOUD.2015.129.

[38] A. Richter, C. Herber, T. Wild, and A. Herkersdorf. Resolving performance interference in sr-iov setups with pcie quality-of-service extensions. In *19th EUROMICRO Conference on Digital System Design (DSD'16)*, pages 454–462, Limassol, Cyprus, August 2016. doi:10.1109/DSD.2016.41.

[39] R. P. Goldberg. Architectural principles for virtual computer systems. Technical report, DTIC Document, 1973.

[40] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[41] S. T. King, G. W. Dunlap, and P. M. Chen. Operating system support for virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '03, pages 6–6, Berkeley, CA, USA, 2003. USENIX Association.

[42] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Transactions on Computer Systems (TOCS)*, 30(4):12, 2012.

[43] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium*, volume 1, pages 225–230, 2007.

[44] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[45] J. Hwang, S. Zeng, F. y Wu, and T. Wood. A component-based performance comparison of four hypervisors. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 269–276. IEEE, 2013.

[46] Virtualization: Architectural Considerations and Other Evaluation Criteria. White paper, VMware, Inc., 2005. URL: `http://www.vmware.com/pdf/virtualization_considerations.pdf` [last checked 2017-04-16].

[47] N. Bhatia. Performance Evaluation of Intel EPT Hardware Assist. Technical report, VMware, Inc., 2009. URL: `https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf` [last checked 2017-04-16].

[48] T. Merrifield and H. R. Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 25–35, New York, NY, USA, 2016. ACM. `doi:10.1145/2892242.2892258`.

[49] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3), 2006.

[50] Advanced Micro Devices, Inc. *AMD I/O Virtualization Technology (IOMMU) Specification*, 2.62 edition, February 2015.

[51] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010. `doi:10.1145/1721654.1721672`.

[52] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 25–38, Washington, DC, USA, 2007. IEEE Computer Society. doi:10.1109/PACT.2007.40.

[53] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM. doi:10.1145/1736020.1736036.

[54] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 283–294, June 2011.

[55] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 22:1–22:14, New York, NY, USA, 2011. ACM. doi:10.1145/2038916.2038938.

[56] P. Mell and T. Grance. The NIST definition of cloud computing. *Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg*, 2011.

[57] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[58] G. Heiser. The Motorola Evoke QA4-A Case Study in Mobile Virtualization. White paper, Open Kernel Labs, Inc., July 2009. URL: https://ssrg.nicta.com.au/publications/papers/Heiser_09:WP:evoke.pdf [last checked 2017-04-16].

[59] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The vmware mobile virtualization platform: Is that a hypervisor in your pocket? *SIGOPS Oper. Syst. Rev.*, 44(4):124–135, December 2010. doi:10.1145/1899928.1899945.

[60] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4android: A generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 39–50, New York, NY, USA, 2011. ACM. doi:10.1145/2046614.2046623.

[61] M. Strobl, M. Kucera, A. Foeldi, T. Waas, N. Balbierer, and C. Hilbert. Towards automotive virtualization. In *2013 International Conference on Applied Electronics*, pages 1–6, Sept 2013.

[62] Z. Gu and Q. Zhao. A state-of-the-art survey on real-time issues in embedded systems virtualization. *Journal of Software Engineering and Applications*, pages 277–290, 2012. `doi:10.4236/jsea.2012.54033`.

[63] S. Xi, J. Wilson, C. Lu, and C. Gill. Rt-xen: Towards real-time hypervisor scheduling in xen. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 39–48, Oct 2011.

[64] I. Ahmad, J. M. Anderson, A. M. Holler, R. Kambo, and V. Makhija. An analysis of disk performance in vmware esx server virtual machines. In *2003 IEEE International Conference on Communications (Cat. No.03CH37441)*, pages 65–76, Oct 2003. `doi:10.1109/WWC.2003.1249058`.

[65] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGOPS Oper. Syst. Rev.*, 40(5):2–13, October 2006. `doi:10.1145/1168917.1168860`.

[66] A. Landau, M. Ben-Yehuda, and A. Gordon. Splitx: Split guest/hypervisor execution on multi-core. In *Proceedings of the 3rd Conference on I/O Virtualization*, WIOV'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[67] K. Fraser, S. H, R. Neugebauer, I. Pratt, and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *In Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[68] R. Russell. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. `doi:10.1145/1400097.1400108`.

[69] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 13–23, New York, NY, USA, 2005. ACM. `doi:10.1145/1064979.1064984`.

[70] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in xen. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.

[71] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[72] S. Chinni and R. Hiremane. Virtual machine device queues. White paper, Intel Corporation, September 2007.

[73] H. Raj and K. Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC '07, pages 179–188, New York, NY, USA, 2007. ACM. `doi:10.1145/1272366.1272390`.

[74] J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, W. Zwaenepoel, and P. Willmann. Concurrent direct network access for virtual machine monitors. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 306–317, Feb 2007. `doi:10.1109/HPCA.2007.346208`.

[75] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10 gb/s using safe and transparent network interface virtualization. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 61–70, New York, NY, USA, 2009. ACM. `doi:10.1145/1508293.1508303`.

[76] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical report, IBM Research Report H-0263, 2008.

[77] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. Van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing iommus for virtualization in linux and xen. In *OLS'06: The 2006 Ottawa Linux Symposium*, pages 71–86, 2006.

[78] PCI-SIG. *PCI Express Base Specification Revision 3.0*, November 2010.

[79] PCI-SIG. *PCI Local Bus Specification Revision 3.0*, February 2004.

[80] PCI-SIG. *PCI-X Addendum to the PCI Local Bus Specification, Revision 2.0*, July 2002.

[81] PCI-SIG. *Single Root I/O Virtualization and Sharing Specification, Revision 1.1*, January 2010.

[82] PCI-SIG. *Multi-Root I/O Virtualization and Sharing Specification, Revision 1.0*, May 2008.

[83] J. Jose, M. Li, X. Lu, K. C. Kandalla, M. D. Arnold, and D. K. Panda. Sr-iov support for virtualization on infiniband clusters: Early experience. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 385–392, May 2013. `doi:10.1109/CCGrid.2013.76`.

[84] NVM Express, Inc. *NVM Express Specification, Revision 1.2.1*, June 2016.

[85] Z. Ding. I/O Virtualization in Enterprise SSDs. In *Storage Developer Conference*, Santa Clara, California, USA, 2015.

[86] T. Wong. AMD Multiuser GPU: Hardware-enabled GPU virtualization for a true workstation experience. White paper, Advanced Micro Devices, Inc., January 2016.

[87] Amazon AWS Documentation - Enhanced Networking on Linux [online]. 2017. URL: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html` [last checked 2017-04-16].

[88] A. Hawley and Y. Eilat. Oracle Exalogic Elastic Cloud: Advanced I/O Virtualization Architecture for Consolidating High-Performance Workloads. White paper, Oracle Corporation, November 2012.

[89] M. Musleh, V. Pai, J. P. Walters, A. Younge, and S. Crago. Bridging the virtualization performance gap for hpc using sr-iov for infiniband. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 627–635, June 2014. `doi:10.1109/CLOUD.2014.89`.

[90] G. K. Lockwood, M. Tatineni, and R. Wagner. Sr-iov: Performance benefits for virtualized interconnects. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, XSEDE '14, pages 47:1–47:7, New York, NY, USA, 2014. ACM. `doi:10.1145/2616498.2616537`.

[91] C. Herber, A. Richter, H. Rauchfuss, and A. Herkersdorf. Self-virtualized can controller for multi-core processors in real-time applications. In *ARCS 2013 – International Conference on Architecture of Computing Systems*, pages 244–255, Prague, Czech Republic, February 2013. Springer Berlin Heidelberg. `doi:10.1007/978-3-642-36424-2_21`.

[92] D. V. Vu, O. Sander, T. Sandmann, S. Baehr, J. Heidelberger, and J. Becker. Enabling partial reconfiguration for coprocessors in mixed criticality multicore systems using pci express single-root i/o virtualization. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6, Dec 2014. `doi:10.1109/ReConFig.2014.7032516`.

[93] O. Sander, S. Baehr, E. Luebbers, T. Sandmann, V. V. Duy, and J. Becker. A flexible interface architecture for reconfigurable coprocessors in embedded multicore systems using pcie single-root i/o virtualization. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 223–226, Dec 2014. `doi:10.1109/FPT.2014.7082780`.

[94] D. V. Vu, O. Sander, T. Sandmann, J. Heidelberger, S. Baehr, and J. Becker. On-demand reconfiguration for coprocessors in mixed criticality multicore systems. In *2015 International Conference on High Performance Computing Simulation (HPCS)*, pages 569–576, July 2015. `doi:10.1109/HPCSim.2015.7237094`.

[95] D. V. Vu, T. Sandmann, S. Baehr, O. Sander, and J. Becker. Virtualization support for fpga-based coprocessors connected via pci express to an intel multicore

platform. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 305–310, May 2014. doi:10.1109/IPDPSW.2014.42.

[96] D. Muench, O. Isfort, K. Mueller, M. Paulitsch, and A. Herkersdorf. Hardware-based i/o virtualization for mixed criticality real-time systems using pcie sr-iov. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 706–713, Dec 2013. doi:10.1109/CSE.2013.109.

[97] D. Muench, M. Paulitsch, and A. Herkersdorf. Iompu: Spatial separation for hardware-based i/o virtualization for mixed-criticality embedded real-time systems using non-transparent bridges. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 1037–1044, Aug 2015. doi:10.1109/HPCC-CSS-ICESS.2015.221.

[98] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. Eli: Bare-metal performance for i/o virtualization. *SIGPLAN Not.*, 47(4):411–422, March 2012. doi:10.1145/2248487.2151020.

[99] Z. Huang, R. Ma, J. Li, Z. Chang, and H. Guan. Adaptive and scalable optimizations for high performance sr-iov. In *2012 IEEE International Conference on Cluster Computing*, pages 459–467, Sept 2012. doi:10.1109/CLUSTER.2012.28.

[100] H. Guan, Y. Dong, K. Tian, and J. Li. Sr-iov based network interrupt-free virtualization with event based polling. *IEEE Journal on Selected Areas in Communications*, 31(12):2596–2609, December 2013. doi:10.1109/JSAC.2013.131202.

[101] Y. Dong, Y. Chen, Z. Pan, J. Dai, and Y. Jiang. Renic: Architectural extension to sr-iov i/o virtualization for efficient replication. *ACM Trans. Archit. Code Optim.*, 8(4):40:1–40:22, January 2012. doi:10.1145/2086696.2086719.

[102] W. L. Guay, S.-A. Reinemo, B. D. Johnsen, C.-H. Yen, T. Skeie, O. Lysne, and O. Tørudbakken. Early experiences with live migration of sr-iov enabled infiniband. *Journal of Parallel and Distributed Computing*, 78:39 – 52, 2015. doi:10.1016/j.jpdc.2015.01.004.

[103] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for Linux VM. In *OLS'08: The 2008 Ottawa Linux Symposium*, pages 261–268, 2008.

[104] A. Kadav and M. M. Swift. Live migration of direct-access devices. *ACM SIGOPS Operating Systems Review*, 43(3):95–104, 2009.

[105] Z. Pan, Y. Dong, Y. Chen, L. Zhang, and Z. Zhang. Compsc: Live migration with pass-through devices. *SIGPLAN Not.*, 47(7):109–120, March 2012. doi:10.1145/2365864.2151040.

[106] X. Xu and B. Davda. Srvm: Hypervisor support for live migration with passthrough sr-iov network devices. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 65–77, New York, NY, USA, 2016. ACM. doi:10.1145/2892242.2892256.

[107] Y. Dong, Z. Yu, and G. Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Proceedings of the First Conference on I/O Virtualization*, WIOV'08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association.

[108] Intel Corporation. *Intel 82576EB Gigabit Ethernet Controller Datasheet*, 2011.

[109] Intel Corporation. *Intel Ethernet Controller I350 Datasheet*, 2.4 edition, 2016.

[110] C. Herber, A. Richter, H. Rauchfuss, and A. Herkersdorf. Spatial and temporal isolation of virtual can controllers. In *Workshop on Virtualization for Real-Time Embedded Systems (VtRES)*, Taipei, Taiwan, August 2013.

[111] W. Jing. Performance isolation for mixed criticality real-time system on multicore with xen hypervisor. Master's thesis, Uppsala University, Department of Information Technology, 2013.

[112] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118, July 2014. doi:10.1109/ECRTS.2014.20.

[113] E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time i/o management system with cots peripherals. *IEEE Transactions on Computers*, 62(1):45–58, Jan 2013. doi:10.1109/TC.2011.202.

[114] D. Muench, M. Paulitsch, and A. Herkersdorf. Temporal separation for hardware-based i/o virtualization for mixed-criticality embedded real-time systems using pcie sr-iov. In *ARCS 2014; 2014 Workshop Proceedings on Architecture of Computing Systems*, pages 1–7, Feb 2014.

[115] P. Mochel. The sysfs filesystem. In *Proceedings of the Linux Symposium*, volume 1, page 313, Ottawa, Ontario, Canada, July 2005.

[116] Automotive, Railway and Avionics Multicore Systems (ARAMiS) [online]. 2011. URL: http://www.projekt-aramis.de/ [last checked 2017-04-16].

[117] Intel Corporation. *Intel 7 Series / C216 Chipset Family Platform Controller Hub (PCH) Datasheet, Revision 003*, June 2012.

[118] G. Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. White paper, Intel Corporation, September 2010.

[119] Z. Amsden. *Timekeeping Virtualization for X86-Based Architectures*, 2010. URL: `https://www.kernel.org/doc/Documentation/virtual/kvm/timekeeping.txt` [last checked 2017-04-16].

[120] VMware, Inc. *Timekeeping in VMware Virtual Machines*, 2011. URL: `http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf` [last checked 2017-04-16].

[121] R. Jones. *Care and Feeding of Netperf*. Hewlett-Packard Company, 2013.

[122] Xilinx Corporation. *Spartan-6 FPGA Integrated Endpoint Block for PCI Express*, October 2010.

[123] International Business Machines Corporation. *POWER8 Processor User's Manual for the Single-Chip Module*, 1.3 edition, March 2016.

[124] International Business Machines Corporation. *ARM Cortex-A72 MPCore Processor Technical Reference Manual*, r0p1 edition, February 2015.

[125] P. Irelan and S. Kuo. Performance Monitoring Unit Sharing Guide. White paper, Intel Corporation, 2009.

[126] Intel Corporation. *Intel Ethernet Controller XL710 Datasheet*, 2.5 edition, 2016.

[127] A. Marletta. cpulimit [online]. URL: `https://github.com/opsengine/cpulimit` [last checked 2017-04-16].

[128] H. J. Koch. Userspace i/o drivers in a realtime context. In *The 13th Realtime Linux Workshop*, 2011.

[129] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[130] PCI-SIG. *PCI Express Base Specification Revision 1.0*, April 2002.

[131] PCI-SIG. *Single Root I/O Virtualization and Sharing Specification, Revision 1.0*, September 2007.

[132] J. Aynsley. *OSCI TLM-2.0 language reference manual*, 2009.

[133] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, June 2015.

A Approximating Packet Processing Times of COTS PCIe Devices

For a CPU and PCIe I/O device combination that is prone to Denial-of-Service (DoS) attacks, which means that the CPU can send PCIe packets faster than the I/O device can process them, it is possible to implement a lean measurement routine that helps approximating packet processing times of the I/O device. The idea is to execute a controlled DoS attack process from the host with a known, fixed amount of PCIe packets, while the rest of the system is idle. If the packet count is chosen high enough, the system will run through three phases, which are depicted in Figure A.1.

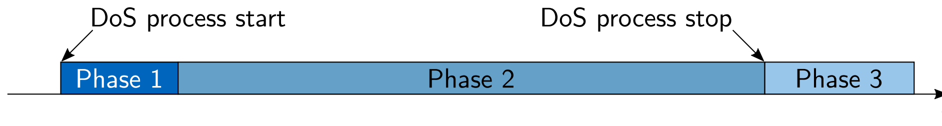


Figure A.1: Three phases of a DoS attack.

Phase 1 – Congestion buildup: The CPU process starts execution of the DoS attack code. At the beginning, buffer resources on CPU, interconnect and I/O device are empty, so that the CPU can send PCIe packets back-to-back at maximum speed. Despite constantly processing packets on the I/O device end, all buffers will fill up eventually, because the CPU is faster.

Phase 2 – Continuous congestion: Eventually, all buffer space on the interconnect will be occupied, and the DoS attack will be in "saturation". In this phase, the CPU is blocked from inserting new PCIe packets in the downstream direction. The next packet can only be released after the I/O device has processed a pending packet. During this phase, packet delay that is normally introduced due to transmission on physical wires and switching overhead is eliminated, because the next packet that must be processed by the I/O device is already waiting in its ingress buffers. By implication, this means that the CPU process (the DoS attack code) that generates the packets is blocked for exactly the time the I/O device needs to process a packet. Hence, during the phase of continuous congestion, the number of issued packets n_{packets} and the processing time of a single packet in the I/O device $t_{\text{proc,I/O}}$ can be used to calculate the time T_{Phase2} as follows:

$$T_{\text{Phase2}} = n_{\text{packets}} \cdot t_{\text{proc,I/O}} \tag{A.1}$$

Phase 3 – Congestion decay: After the CPU placed the last of its fixed amount of PCIe packets in its downstream buffers, process execution of the DoS attack terminates. The congestion on the interconnect will slowly decay until the last packet is processed. This phase can be neglected for packet processing time approximation.

The overall execution time of the DoS process is therefore $T_{\text{Phase1}} + T_{\text{Phase2}}$, which can be measured by the operating system. Unfortunately, there is no straightforward way of measuring T_{Phase1} or T_{Phase2} individually. However, if n_{packets} is chosen large enough so that $T_{\text{Phase2}} >> T_{\text{Phase1}}$, it is possible to get a good approximation of $t_{\text{proc,I/O}}$ by simply calculating

$$t_{\text{proc,I/O}} \approx \frac{T_{\text{Phase1}} + T_{\text{Phase2}}}{n_{\text{packets}}}. \tag{A.2}$$

In the following, two examples for the host system that was presented in Section 4.3.1 are given. In the first example, 30_000_000 PCIe packets with 64 bit payloads are sent to the first address of the first BAR of the 82576 NIC's address space. This number was found to be high enough so that Phase 1 had no significant impact anymore. The time to execute the attacker process was measured with the standard Unix `time` tool. Output of the controlled DoS process and `time` is given as follows:

```
target:          /sys/bus/pci/devices/0000:08:10.1/resource0
offset:          0
loops:           30000000
target BAR size: 16 KiB
target addr:     0xBAA81000
13.21 user 0.00 system 0:13.21 elapsed 100% CPU
```

Overall, the controlled DoS process took 13.21 s to execute. Hence, the time for processing a single packet approximately equals

$$t_{\text{proc,I/O}} \approx \frac{13.21\,\text{s}}{30000000} \approx 440\,\text{ns}. \tag{A.3}$$

In the second example, the offset into the device's BAR is set to 10240, therefore addressing a different memory location within the I/O device than in the first example. Output was as follows:

```
target:          /sys/bus/pci/devices/0000:08:10.1/resource0
offset:          10240
loops:           30000000
target BAR size: 16 KiB
target addr:     0xFC524800
16.01 user 0.00 system 0:16.01 elapsed 100% CPU
```

This time, the processing time approximates

$$t_{\text{proc,I/O}} \approx \frac{16.01\,\text{s}}{30000000} \approx 534\,\text{ns}. \tag{A.4}$$

In conclusion, the memory address that was tested with the second example is better suited for a DoS attack than the first one. The longer processing times would result in a more severe impact of a DoS attack, because during congestion, more time passes until a victim VM can insert a PCIe packet on the interconnect.