Walk-In : flexible virtualization APIs for performance and security in the modern datacenter

Tu Dinh Ngoc

To cite this version:

Tu Dinh Ngoc. Walk-In : flexible virtualization APIs for performance and security in the modern datacenter. Library and information sciences. Université de Toulouse, 2024. English. NNT : 2024TLSES002 . tel-04579334

HAL Id: tel-04579334
https://theses.hal.science/tel-04579334

Submitted on 17 May 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Université de Toulouse

Doctorat de l'Université de Toulouse

préparé à l'Université Toulouse III - Paul Sabatier

Walk-In : interfaces de virtualisation flexibles pour la performance et la sécurité dans les datacenters modernes

---

Thèse présentée et soutenue, le 5 février 2024 par

Ngoc Tu DINH

École doctorale
EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse

Spécialité
Informatique et Télécommunications

Unité de recherche
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Georges DA COSTA et Daniel HAGIMONT

Composition du jury
Mme Julia LAWALL, Rapporteure, INRIA Paris
M. David BROMBERG, Rapporteur, Université de Rennes
M. Renaud LACHAIZE, Examinateur, Université Grenoble Alpes
M. Georges DA COSTA, Directeur de thèse, Université Toulouse III - Paul Sabatier
M. Daniel HAGIMONT, Co-directeur de thèse, Toulouse INP
M. Noel DE PALMA, Président, Université Grenoble Alpes

Membres invités
M. Boris TEABE, Toulouse INP

Doctorate from the University of Toulouse

prepared at the Université Toulouse III - Paul Sabatier

Walk-In: Flexible Virtualization APIs for Performance and Security in the Modern Datacenter

---

Thesis presented and defended on 05 Feb 2024 by:

Tu Dinh Ngoc

Doctoral school:
EDMITT - Ecole Doctorale Mathématiques, Informatique, Télécommunications de Toulouse

Specialty:
Computing and Telecommunications

Research unit:
IRIT : Institut de Recherche en Informatique de Toulouse

Thesis directed by:
Georges Da Costa and Daniel Hagimont

Jury composition:
Ms. Julia Lawall, Reviewer, INRIA Paris
Mr. David Bromberg, Reviewer, Université de Rennes
Mr. Renaud Lachaize, Examiner, Université Grenoble Alpes
Mr. Georges Da Costa, Thesis director, Université Toulouse III - Paul Sabatier
Mr. Daniel Hagimont, Thesis director, Toulouse INP
Mr. Noel De Palma, Jury president, Université Grenoble Alpes

Guest members:
Mr. Boris Teabe, Toulouse INP

Abstract

Virtualization is a powerful tool that brings numerous benefits for the security, efficiency and management of computer systems. Modern infrastructure therefore makes heavy use of virtualization in almost every software component. However, the extra hardware and software layers present various challenges to the system operator.

In this work, we analyze and identify the challenges relevant to virtualization. Firstly, we observe a complexification of maintenance from the numerous software layers that must be constantly updated. Secondly, we notice a lack of transparency on details of the underlying infrastructure from virtualization. Thirdly, virtualization has a damaging effect on system performance, stemming from how the layers of virtualization have to be navigated during operation.

We explore three approaches of solving the challenges of virtualization through adding flexibility into the virtualization stack.

- Our first contribution tackles the issue of maintainability and security of virtual machine platforms caused by the need to keep these platforms up-to-date. We introduce HyperTP, a framework based on the hypervisor transplant concept for updating hypervisors and mitigating vulnerabilities.
- Our second contribution focuses on performance loss resulting from the lack of visibility of non-uniform memory access (NUMA) topologies on virtual machines. We thoroughly evaluate I/O workloads on virtual machines running on NUMA architectures, and implement a unified hypervisor-VM resource allocation strategy for optimizing virtual I/O on said architectures.
- For our third work, we focus our attention on high-performance storage subsystems for virtualization purposes. We present NVMetro, a flexible yet easy to use virtual storage platform that supports the implementation of fast yet efficient storage functions.

Together, our solutions demonstrate the tradeoffs present in the configuration spaces of virtual machine deployments, as well as how to reduce virtualization overhead through dynamic adjustment of these configurations.

Résumé en français

La virtualisation est un outil puissant conférant de nombreuses avantages pour la sécurité, l'efficacité et la gestion des systèmes informatiques. Par conséquent, la virtualisation est largement utilisée dans les infrastructures informatiques modernes. Cependant, les couches materielles et logicielles supplémentaires posent de nouveaux défis aux administrateurs systèmes.

Dans cette thèse, nous présentons notre analyse des défis liés à la virtualisation. Nous constatons que les nombreux composants de virtualisation doivent être tenus à jour, ce qui complique fortement la maintenance logicielle. Deuxièmement, nous remarquons que la virtualisation masque les spécificités du matériel. Enfin, elle réduit les performances des systèmes du fait de la coordination entre les logiciels impliqués par la virtualisation.

Nous étudions trois approches pour relever les défis présentés ci-dessus en augmentant la flexibilité de la pile de logiciels de virtualisation.

- Notre première contribution concerne la maintenabilité et la sécurité des plateformes de machines virtuelles en tenant compte du besoin de les maintenir à jour. Nous présentons notre cadre « HyperTP » pour la mise à jour des hyperviseurs et l'attenuation des vulnérabilités pendant le délai entre leur découverte et leur correction.
- Notre deuxième contribution concerne la perte de performance résultant du manque de visibilité des topologies NUMA dans les machines virtuelles. Nous étudions la performance des charges de travail d'E/S virtuelles afin de déterminer une politique d'allocation des ressources pour l'optimisation des E/S.
- Notre troisième contribution concerne la virtualisation des systèmes de stockage à haute performance. Nous présentons la plateforme de stockage NVMetro, qui permet la réalisation de fonctions de stockage performantes grâce à sa flexibilité et à sa facilité d'utilisation.

En résumé, nos contributions montrent les compromis présents dans les configurations des machines virtuelles, ainsi que la réduction des coûts de virtualisation par la manipulation dynamique de ces configurations.

Acknowledgements

Firstly, I would like to acknowledge my PhD jury members for the time and effort you have spent on my thesis. I would like to thank Ms. Julia Lawall and Mr. David Bromberg for agreeing to review my thesis and giving me helpful comments and feedback; and Mr. Noel De Palma and Mr. Renaud Lachaize for examining my work and joining my thesis defense. I greatly enjoyed our discussion as well as your deep insights.

I am very grateful to Mr. Daniel Hagimont and Mr. Georges Da Costa for guiding me through my PhD. You gave me the incredible opportunity to do research at IRIT, and your advices and wisdom were truly valuable to my project.

A special thank goes to Mr. Boris Teabe, who has assisted me throughout my years in France. You went above and beyond in accompanying me both at and outside of work. I am very glad to have you as a mentor and excellent collaborator.

I would like to also thank the laboratory secretaries Ms. Sylvie Armengaud and Ms. Vanessa Adjeroud for your hospitality and support during my PhD. You were an invaluable help to all of us at IRIT.

Thanks to all of my good friends in France: Armel, Jean-Baptiste, Boris Wembe, Djob, Firmin, Kevin, Hiep, Hoang, and so many others, who were a great fun to talk with every day, and made the office a truly enjoyable place to be at.

Lastly, I cannot forget to thank my father, mother and sister, who cared about and encouraged me when I was far away from home.

Publications

Following is a list of works that have been accomplished as part of my research project. Works **2**, **3** and **4** constitute parts of this thesis.

First-author works:

1. Tu Dinh Ngoc, Boris Teabe, Alain Tchana, Gilles Muller, and Daniel Hagimont. Mitigating vulnerability windows with hypervisor transplant. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 162–177, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383349. doi:10.1145/3447786.3456235

2. Tu Dinh Ngoc, Boris Teabe, Daniel Hagimont, and Georges Da Costa. Optimized resource allocation on virtualized non-uniform I/O architectures. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 432–441. IEEE, 2022. doi:10.1109/CCGrid54584.2022.00053

3. Tu Dinh Ngoc, Boris Teabe, Alain Tchana, Gilles Muller, and Daniel Hagimont. HyperTP: A unified approach for live hypervisor replacement in datacenters. *Journal of Parallel and Distributed Computing*, page 104733, 2023. doi:10.1016/j.jpdc.2023.104733

4. Tu Dinh Ngoc, Boris Teabe, Georges Da Costa, and Daniel Hagimont. Flexible NVMe request routing for virtual machines. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024 **[To appear]**

Other works:

5. Brice Ekane, Tu Dinh Ngoc, Boris Teabe, Daniel Hagimont, and Noel De Palma. FlexVF: Adaptive network device services in a virtualized environment. *Future Generation Computer Systems*, 127:14–22, 2022. doi:10.1016/j.future.2021.08.011

6. Jean-Baptiste Decourcelle, Tu Dinh Ngoc, Boris Teabe, and Daniel Hagimont. Fast VM replication on heterogeneous hypervisors for robust fault tolerance. In *Proceedings of the 24th International Middleware Conference*, Middleware '23, pages 15–28, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701771. doi:10.1145/3590140.3592849

Contents

1 Introduction 1

1.1 Overview . . . 1

1.2 The challenges of virtualization . . . 2

1.3 Our contributions . . . 2

1.4 Structure of this thesis . . . 4

2 Context 5

2.1 The various facets of virtualization . . . 5

2.2 Inside platform virtualization . . . 6

2.3 NUMA and I/O virtualization . . . 12

2.4 The NVM Express specification . . . 13

2.5 Linux's eBPF engine . . . 14

3 HyperTP: A unified approach for live hypervisor replacement 15

3.1 Overview . . . 16

3.2 Design of HyperTP . . . 18

3.3 Prototype . . . 23

3.4 Evaluation . . . 29

3.5 Discussion . . . 40

3.6 Related works . . . 41

3.7 Summary . . . 43

4 Optimized resource allocation on virtualized non-uniform I/O architectures 45

4.1 Overview . . . 46

4.2 Impact of NUIOA on VM performance . . . 47

4.3 NUIOA-aware VM resource allocation strategy . . . 53

4.4 Evaluation . . . 55

4.5 Related works . . . 61

4.6 Summary . . . 62

5 NVMetro: Enhancing mediated NVMe virtualization with eBPF 65

5.1 Overview . . . 66

5.2 Design of NVMetro . . . 67

5.3 Use cases . . . 72

5.4 Evaluation . . . 77

5.5 Related works 86
5.6 Summary 88

6 Conclusion and perspectives **89**

Chapter 1

Introduction

Contents

1.1 Overview 1
1.2 The challenges of virtualization 2
1.3 Our contributions 2
1.4 Structure of this thesis 4

1.1 Overview

Among other advances, virtualization has been the main driver of changes in how organizations develop and deploy software. Numerous hardware and software tools have been developed for the purposes of virtualization: specialized CPU instructions, hardware protocols, software stacks (e.g. VMware vSphere, Kubernetes), among others.

At its core, virtualization entails providing an abstract representation of computing resources on top of real ones. Almost any resource can be virtualized: memory, storage/networking, operating system environments, etc. Virtualization brings various benefits to the operation of computer systems: firstly, virtualization helps increases hosting density by sharing the same physical hardware across multiple execution environments. Secondly, virtualization provides a layer of isolation for security, reliability and resource control reasons. Lastly, it lets the system operator freely manipulate virtual resources: granting and taking them away, saving and loading their states, or even migrating them across machines.

The modern deployment model therefore contains multiple instances of virtualization. Consider an example of a function-as-a-service (FaaS) deployment, where virtualization-adjacent components are shown in bold: a serverless function runs in a **container** hosted in a **virtual machine** (VM) managed by a **hypervisor**. The function communicates with the outside world through a **container network interface** provided by an **overlay network** backed by the virtual machine's **virtual network interface**.

1.2 The challenges of virtualization

As we can see, modern software deployments make extensive use of virtualization in various points of their infrastructure. However, such an extensive virtualization stack brings significant challenges to the system operator. We list below a few examples of challenges we tackle in this thesis.

Manageability. Each of these virtualization layers include their own software stack which needs to be maintained, updated and secured. Particularly, updating a virtualized infrastructure is critical to maintaining its security, yet there exists numerous difficulties in discovering, testing and deploying patches [7, 8]. Zhang et al. reports needing a total of 15 days just to upgrade a virtual machine monitor to a newer version [9]. Together, these tasks take up a considerable amount of time and resources, without eliminating the *vulnerability window* between the moment a vulnerability becomes known, and when the infrastructure is fully secured (also known as a *patch gap*).

Transparency. Virtualization hides details of the underlying infrastructure from the application. While this is often intentionally done for security and portability purposes, it also takes away the opportunity to optimize workloads for the systems they are running from. In particular, modern servers are *distributed systems* consisting of islands of compute, memory and I/O linked together with interconnects. While solutions exist to reflect this architecture to VMs, the information they provide is often not complete, and the reconfiguration inefficient [10, 11].

Performance. The enormous software stack of modern deployments comes with a significant performance cost, as more work is needed to mediate an application's operations. Li et al. demonstrates an overhead of over 3× for device emulation in Xen, a common part of virtual machines running on said platform [12]. Zhong et al. shows that software has become the bottleneck of high-performance storage, with the file system driver, kernel crossing and block layer operations accounting for over 40% of average I/O latency [13].

1.3 Our contributions

In this thesis nicknamed "Walk-In", we propose to mitigate the above challenges through adding *flexibility* in the virtualization software stack. The core idea of our work is to explore the configuration space of virtualized deployments, and choose an appropriate configuration that targets a given challenge. By presenting multiple choices of virtualization methods while deeply understanding the pros and cons of each method, our work aims to solve the inefficiencies of virtualized deployments by choosing a virtualization method that best fits each use case. We apply our idea of flexibility in each of the challenges presented above:

1. We tackle the problems of hypervisor updating and vulnerability mitigation by dynamically switching from a current hypervisor to another. By introducing two levels of flexibility in hypervisor implementation and switching method, we can effectively *avoid* an active vulnerability in minimal time and with minimal overhead.

2. By analyzing the various tradeoffs involved in I/O performance, as well as VM and application configurations, we create an I/O-aware resource allocation strategy that optimizes the performance of virtual I/O on non-uniform memory access (NUMA) architectures.

3. We present a virtual storage platform that combines an intelligent request classifier and router mechanism with multiple I/O paths for implementing sophisticated storage functions while maintaining high performance and security.

For our first contribution, we build *HyperTP*, a generic hypervisor replacement framework which flexibly combines two approaches: in-place server micro-reboot-based hypervisor transplant (noted *InPlaceTP*) and live VM migration-based hypervisor transplant (noted *MigrationTP*). HyperTP hinges on a *VM state hierarchy* for organizing different types of hypervisor memory states in terms of their relation to VM execution, and an *Unified Intermediate State Representation* (UISR) that abstracts VM-relevant memory states between multiple different hypervisors. We describe our implementations of both approaches, including technical details of our UISR design and the transplant process. Our evaluation results show that HyperTP delivers satisfactory performance: (1) InPlaceTP imposes minimal VM downtime, even under increasing number of VMs and memory sizes; (2) MigrationTP changes a VM's underlying hypervisor while taking the same time and impacting virtual machines with the same performance degradation as normal live migration. Finally, we discuss how the combination of InPlaceTP and MigrationTP can be used to address the challenges of upgrading a hypervisor cluster, and to mitigate known unpatched hypervisor vulnerabilities.

Secondly, we investigate the impact of virtualized I/O on NUMA architectures. As I/O devices are typically connected to one particular NUMA node, this leads to a situation where device access on one node is faster than another. This phenomenon is called *non-uniform I/O access* (NUIOA). This non-uniformity impacts the performance of I/O applications that are not executed on the correct NUMA node. Our contribution in this chapter is twofold: (1) we thoroughly study the impact of NUIOA on application performance in VMs; and (2) we propose a VM resource allocation strategy that reduces the impact of NUIOA by splitting home node resources across all VMs. We implement our allocation strategy on the Xen hypervisor and carry out evaluations with well-known benchmarks to validate our strategy. The obtained results show that with our NUIOA allocation scheme, we can improve the performance of application in VMs by up to 20% compared to common allocation strategies.

With our third idea, we first observe that existing storage virtualization tools either depend on a heavy and inefficient I/O stack that is not optimized for parallelism, or require a separate API that is difficult to manage and monitor. We introduce *NVMetro*, a

solution based on the NVMe protocol that proposes a flexible choice between multiple I/O paths to ease the development of adaptive and performant virtual storage. NVMetro provides two components: (1) an intelligent I/O classification and routing framework powered by Linux's *Extended Berkeley Packet Filter* (eBPF); and (2) an easy-to-use and performant API to assist the creation of *userspace I/O functions* (UIFs) within our framework. We demonstrate the benefits of NVMetro by implementing two virtual storage functions, and we evaluate them using various benchmarks. The obtained results show that NVMetro achieves a performance and scalability comparable to bleeding-edge, kernel-bypass technologies while retaining the flexibility of traditional OS-based storage APIs.

1.4 Structure of this thesis

In Chapter 2, we introduce the various aspects of platform virtualization, including the components of a virtual machine, their implementation in modern hypervisors, and the various tradeoffs of different VM platforms. We also investigate the NVM Express specification and Linux's eBPF engine, important components of high-performance I/O stacks. In Chapter 3, we present HyperTP through the lens of our hypervisor transplant concept, with detailed descriptions of its design and implementation, as well as relevant discussions of its security and performance properties. Chapter 4 evaluates I/O workloads under multiple scenarios to provide our observations on the effects of NUIOA on VM performance and to offer a strategy for improving performance in these scenarios. In Chapter 5, we present NVMetro from its design criteria, describe and evaluate several relevant use cases, and compare our solution to competing ones. Finally, Chapter 6 summarizes Walk-In's common theme of *flexibility*, while giving perspectives about future works.

Chapter 2

Context

Contents

2.1 The various facets of virtualization . . . 5

2.2 Inside platform virtualization . . . 6

2.2.1 Mechanisms of machine component virtualization . . . 7

2.2.2 Virtualization of the x86 architecture . . . 8

2.2.3 Memory virtualization . . . 8

2.2.4 Virtualization of I/O devices . . . 9

2.2.5 Virtualizing signal delivery . . . 10

2.2.6 Managing virtual machine states . . . 11

2.3 NUMA and I/O virtualization . . . 12

2.4 The NVM Express specification . . . 13

2.5 Linux's eBPF engine . . . 14

2.1 The various facets of virtualization

As stated in Chapter 1, virtualization aims to partition a *physical resource* into one or more instances of a corresponding *virtual resource*. To further clarify this concept, we provide below a list of virtual resources commonly available in modern software architectures:

- *Virtual memory* serves as the main backing principle of OS processes. With virtual memory, each process gains a separate view of its own memory, isolated from other processes on the same system. Virtual memory functionalities are backed by *memory management units* available on modern CPUs, which translate *virtual addresses* belonging to different *virtual address spaces* into *physical addresses* for accessing main memory and I/O devices.

- *Virtual machines* (VMs) supply an operating system environment that acts like a physical computer, complete with virtual CPUs, memory, I/O, and so on. Managed by a *virtual machine monitor* (VMM) or *hypervisor*, VMs provide a robust isolation of workloads and administrative duties. Thanks to a familiar and diverse choice of guest operating systems, VMs form one of cloud computing's core offerings, *infrastructure-as-a-service* (IaaS).

- *Containers* are a partitioning of OS resources (processes, filesystems, network devices) into separate domains. Containers are managed by a *container runtime*, and may be backed by a single OS (e.g. Linux, using its various namespaces and control group features) or by separate VMs. Developers create containers from self-contained bundles of applications and their dependencies called *container images*. Container platforms such as Docker and Kata Containers have attracted significant interest in recent years due to their low overhead and high deployment consistency.

2.2 Inside platform virtualization

Platform virtualization refers to the process where *hypervisors* partition a *host* machine to provide an isolated *virtual machine* environment for *guest operating systems*. According to Popek and Goldberg, such an environment invokes the following properties: efficiency, where unprivileged instructions are executed without interception; resource control, where the hypervisor retains complete control over any resources granted to the VM; and equivalence, where the guest software can access resources in the same way as software running on unvirtualized hardware [14].

This thesis focuses specifically on virtual machines and their implementations in common hypervisors (KVM, Xen) on the x86 architecture. To this end, we first break down a virtual machine environment into its components:

- *Virtual CPUs* (vCPUs), which present the guest OS and applications with an execution context. Each vCPU possesses its own states, such as registers, interrupts, execution modes, and so on.

- *VM memory*, in the form of a separate guest physical address space. Like in a physical machine, VM physical addresses can be mapped to main memory or a so-called *memory-mapped I/O device*.

- Various *virtual devices* that provide I/O services to the VM. We consider two classes of virtual devices: critical *platform devices* necessary for a VM's operation, and *bulk I/O devices* that supply the VM with storage, networking, etc. Under our definition, vCPUs are naturally considered a type of virtual device.

The next sections present individual elements of platform virtualization for each of the aforementioned components.

2.2.1 Mechanisms of machine component virtualization

VM components are commonly virtualized in one of three ways: **trap-and-emulate**, where sensitive operations within the guest are intercepted by hardware, then the hypervisor alters the VM's state accordingly; **binary translation**, where the guest software is broken down into its component executable instructions, and any sensitive instructions within the guest program are replaced with innocuous ones that replicate their effects within the VM's context; and **hardware-assisted virtualization**, where a commonly-used resource (vCPU, VM memory) is exposed directly by hardware in an efficient fashion.

Each virtualization method listed above has its own advantages and disadvantages. For example, hardware-assisted virtualization brings the best performance but obviously requires support from the underlying hardware. In contrast, binary translation and trap-and-emulate often come with a significant overhead, especially for high-performance bulk I/O devices such as storage and networking. To mitigate this shortcoming, modern hypervisors optimize virtual I/O by providing a VM-specific I/O interface via special instructions and protocols in a process called *paravirtualization* [15, 16].

Common VM platforms such as KVM and Xen consist of two components: a *hypervisor kernel* that runs at the highest privilege level and provides virtualization primitives; as well as a *deprivileged VMM* (e.g. QEMU, kvmtool, Firecracker) that uses the hypervisor kernel's API to manage VMs and grant them access to resources.[1] This kernel/user split implies a tradeoff between security and performance. To elaborate, such a separation minimizes a VM platform's attack surface by running most of its code in an unprivileged, sandboxable mode subject to the operating system's controls. At the same time, signaling code paths between the VM and virtual device provider become much slower. Events coming from the VM (e.g. control register writes, I/O accesses) must pass through the hypervisor kernel before being relayed to the VMM. Conversely, in order to signal the VM with an interrupt, the VMM must request the hypervisor kernel via a system call [12]. For this reason, hypervisors often directly virtualize performance-critical devices (e.g. interrupt controllers). The component split is also configurable depending on security requirements, where in-kernel virtual devices can be disabled and replaced with VMM-based ones in sensitive environments [17].

Goldberg categorized hypervisors into two types: *type-1 hypervisors* and *type-2 hypervisors* [18]. Type-1 hypervisors (e.g. Xen, Hyper-V) run directly on a physical machine and take full control of its hardware. Type-2 hypervisors (e.g. KVM, VirtualBox) instead work as a component of the host OS. The two types of hypervisors differ in implementation. Under a type-1 hypervisor, system resources (memory, scheduling, device access) are controlled by the hypervisor, and the host OS runs in virtualized mode. In contrast, with a type-2 hypervisor, the host OS runs in unvirtualized mode and retains control over resource allocation, while providing these services to the hypervisor.

[1] In this thesis, we use the term *[userspace] VMM* to refer to the deprivileged component of a virtual machine platform. Conversely, the privileged component is called the *hypervisor [kernel]*.

2.2.2 Virtualization of the x86 architecture

The x86 architecture was not designed for virtualization at the outset and presented significant challenges to virtualization, as it presented several virtualization-sensitive instructions that were also unprivileged, and thus not intercepted by hardware [19]. Initial attempts at x86 virtualization found in VMware and QEMU used a binary translation approach to emulate sensitive instructions in software [19, 20, 21]. However, such an approach remain slow, with the VMM having to handle numerous potential issues [22].

Intel patched the virtualization gap of x86 in introducing their VT-x virtualization technology [22], which introduces separate host *(VMX root)* and guest *(VMX non-root)* execution modes. Each vCPU in VMX non-root mode is associated with a VM control structure (VMCS) containing its state, and which the host can manipulate using `VMREAD` and `VMWRITE` instructions. From VMX root mode, the host executes a guest vCPU through a *VM entry* operation, which puts the corresponding physical CPU in non-root mode. Subsequently, any sensitive instruction executed by the guest either directly manipulates the guest state without affecting the host, or causes a *VM exit* that notifies the host of such an event. Figure 2.1 illustrates the various execution modes provided by VT-x [23].

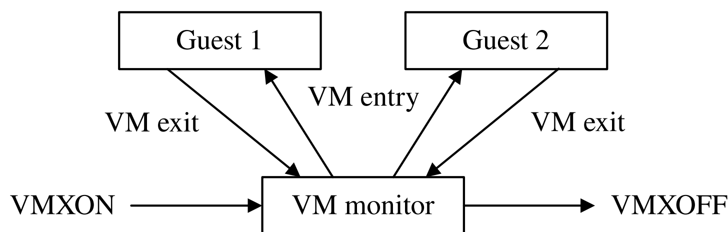

Figure 2.1: Interaction of a VMM and guests [23].

2.2.3 Memory virtualization

Prior to specific memory virtualization capabilities in hardware, x86 memory virtualization is performed through the *shadow paging* technique. In summary, for each process in the guest VM associated with a guest page table (GPT), the hypervisor creates a *shadow page table* (SPT) which translates virtual addresses of said process to host physical addresses. The hypervisor retains exclusive control over address translation by intercepting all memory management operations (CR3 register, TLB flushes). It also intercepts writes to guest page tables and updates the corresponding SPTs in parallel. Upon an attempted task switch from the guest, the hypervisor substitutes the correct SPT. These interceptions cause an execution overhead during these operations, especially with e.g. task creations which heavily involve page table manipulations. Xen improved upon shadow paging by introducing a paravirtualized interface that batches page table updates [16]; however, this paravirtualization does not totally eliminate the overhead of shadow paging.

Bhargava et al. [24] introduced *nested paging* as part of AMD's AMD-V virtualization technology to reduce the overhead of memory virtualization. With this technology, hypervisors create *nested page tables* (NPT) associated to a VM control block (AMD-V's equivalent of VT-x's VMCS). These NPTs transparently translate from guest physical addresses to host physical addresses during guest execution, while letting guests manage their own GPTs without hypervisor interception. However, nested paging causes an overhead in memory accesses due to needing two levels of address translation. This particular overhead of nested paging has been explored in previous works [25, 26]. To compensate, huge pages can be used to reduce the overhead of nested paging by reducing the total number of translation levels, as well as the TLB space needed for caching translations.

Table 2.1 summarizes the behaviors of guest operations for each memory virtualization method.

Table 2.1: Summary of memory virtualization behaviors per guest operation.

G: guest	VA: virtual address	Green : optimal performance
H: host	PA: physical address	Yellow : scope for improvement
S : shadow	PT: page table	Red : poor performance
N: nested		

Method	**Memory accesses**	**GPT edits**	**Context switches**
Shadow paging	$GVA \xrightarrow{SPT} HPA$ (Green)	Trap (Red)	Trap (Red)
Paravirtualized	$GVA \xrightarrow{SPT} HPA$ (Green)	Batched (Yellow)	PV SPT switch (Red)
Nested PT	$GVA \xrightarrow{GPT} GPA \xrightarrow{NPT} HPA$ (Yellow)	Direct (Green)	Direct (Green)

2.2.4 Virtualization of I/O devices

Virtual I/O devices are commonly exposed to the VM using one of the following methods:

- *Emulation*, where the VMM mimics the behavior of a real I/O device, e.g. *Intel 82574L GbE Controller, Realtek RTL8139* (networking); *LSI MegaRAID SAS 1078, SiI3112A PCI to Serial ATA Controller* (storage).[2] Emulated devices often have low performance, since each I/O operation requires several MMIO accesses, each of which must be intercepted by the hypervisor and sent to the VMM.

- *In-VMM paravirtualization*, where the I/O device protocol exposed by the VMM is VM-specific (e.g. by following the *Virtio* specification [27]). The guest OS must contain the appropriate drivers for these protocols.

- *In-hypervisor paravirtualization*, where the hypervisor kernel itself emulates the virtual device instead of the deprivileged VMM. For example, Linux provides *Vhost* modules that implement Virtio network and storage devices directly inside the kernel for better I/O performance.

[2] Virtual device examples provided by QEMU 6.2.

- *Device passthrough*, where an entire physical device is assigned to a VM. As physical devices have access to the system's memory via DMA, their access is often guarded with a *I/O memory management unit* (IOMMU), which constrains their memory access via a virtual input/output address space. Linux provides device passthrough facilities through its *Virtual Function I/O* (VFIO) framework.

As an example, Table 2.2 presents Gregg's [28] breakdown of virtualization methods found in the Xen hypervisor, categorized by the virtual device type.

Table 2.2: Spectrum of virtualization modes in the Xen hypervisor [28].

P : paravirtualized
VS : virtualized in software
VH: virtualized in hardware

Green : optimal performance
Yellow : scope for improvement
Red : poor performance

Type	**Mode**	**With**	**Disk and network**	**Interrupts, timers**	**Emul. MB, legacy boot**	**Priv. instr., page tables**
Fully virtualized	HVM		VS (Red)	VS (Red)	VS (Yellow)	VH (Green)
Hybrid, Xen 3.0	HVM	PV drivers	P (Green)	VS (Red)	VS (Yellow)	VH (Green)
Hybrid, Xen 4.0.1	HVM	PVHVM drivers	P (Green)	P (Green)	VS (Yellow)	VH (Green)
Hybrid, Xen 4.4	PV	HVM ("PVH")	P (Green)	P (Green)	P (Green)	VH (Green)
Fully paravirtualized	PV		P (Green)	P (Green)	P (Green)	P (Yellow)

A common pattern of I/O protocols is shared memory-based communication. This pattern is used in e.g. Xen PV, Virtio and the NVMe protocol, and consists of two parts: a shared memory buffer, often in the form of a producer/consumer ring; and two-way signal delivery mechanisms between the VMM and guest. I/O requests from one side (e.g. the guest sending a packet) are accomplished by first populating a ring entry with data pointers, then sending a signal to inform the other party of the update in ring status.

2.2.5 Virtualizing signal delivery

As mentioned above, for virtualized I/O protocols to function, the VMM must provide VMs with signal delivery in the form of interrupts: timer, device, interprocessor (IPI), etc. Similar to any other devices, interrupt controllers are exposed to the VM in several ways:

- *Emulation:* where a well-known hardware interrupt controller is mimicked in software (e.g. the Intel 8259 PIC);

- *Paravirtualization:* Since efficient interrupt virtualization is crucial to virtual I/O performance, VMMs often provide a paravirtualized interrupt interface with a lower overhead. Namely, Xen and Hyper-V both propose their own paravirtual interrupt mechanism configured via hypercalls. Newer hardware interrupt controller specifications (e.g. x2APIC) are designed to be simpler and more efficient to emulate, and its implementations can perform as well as paravirtualized interrupt controllers.

- *Hardware-assisted:* Newer CPUs from Intel and AMD provide explicit support for virtualizing interrupt controllers as part of their hardware virtualization technologies. These include automatic emulation of several virtual interrupt types (timers, IPIs); emulation of interrupt controller registers; *posted interrupts* for direct delivery of interrupts to vCPUs without VM exits; and the related *interrupt remapping* for delivery of interrupts originating from hardware.

2.2.6 Managing virtual machine states

As part of virtual machine management tasks, VMMs provide various methods for controlling their execution states. Besides the basic start/stop and pause/resume, the following operations are commonly provided:

- *VM state dumping and loading.* With this operation, the VM's running states (CPU registers, memory contents, device states) are dumped to stable storage as a save file with a well-defined format. This save file can later be used to reconstruct the running VM.

- *VM live migration.* This operation moves a running VM from one host to a second host with minimal downtime. Based on the same principles as VM state dumping, live migration is often implemented in one of two ways:

  - *Pre-copy migration:* VM memory is copied to the second host in the background without pausing the VM. Once enough of the memory contents have been copied, the VM is paused and the remaining states are transferred over so that the VM could be resumed in the second host. Since the VM keeps running in the original host during migration, any pages modified by the VM must be copied again, a lengthy process on busy VMs.
  - *Post-copy migration,* where the VM is first transferred over to the second host, but at first without its memory. Any memory accesses will trap to the hypervisor, where it will force the memory pages in question to be copied over; the rest will be copied in background. Post-copy has the disadvantage of severely slowing down the VM during these page faults, as they involve network activity and multiple transfers of control between the VM, VMM and hypervisor.

- *VM snapshotting.* Using this operation, a VM's disk contents (and often its running states) are checkpointed into a stable on-disk snapshot. Any further changes

toward these states are redirected into a differential image. At any time, the VM's states can be rolled back to that of the snapshot.

2.3 NUMA and I/O virtualization

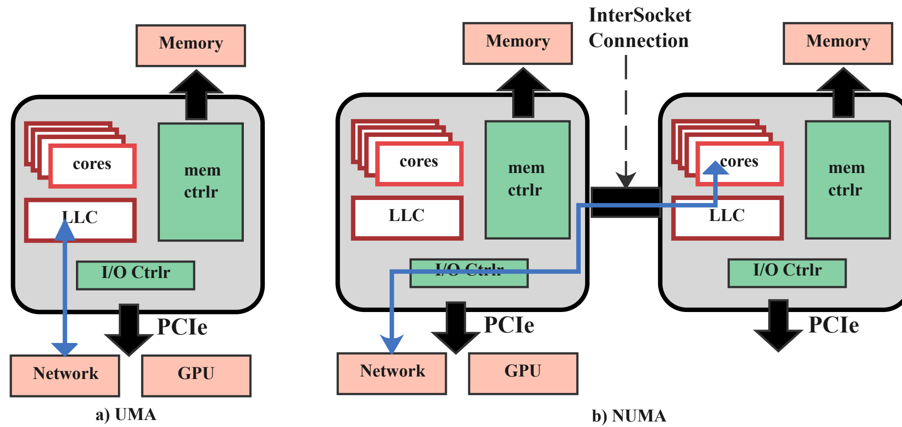

Figure 2.2: I/O in a server based on a (a) UMA (i.e. non-NUMA) architecture; (b) NUMA architecture.

As the demand for computing power increases, manufacturers aim to integrate more and ever faster CPU cores, memory, storage, networking etc. into their server products. Yet with CPUs increasing in speed and parallelism, it becomes more difficult to feed them with data. Signal path length becomes a significant concern, as data latencies are constrained by physical limitations of the hardware itself; hardware designers therefore need to limit the distances between CPUs, memory and peripheral devices.

On architectures where the processor is connected to devices using a single bus link (which is often the case for single-socket systems with uniform memory access, UMA), all CPU cores share the same I/O path to a certain device (blue arrow on Figure 2.2a). However, with ever-increasing demands for processing power and hosting density, current servers can come with multiple processors, each with its own CPU cores, memory hierarchy and I/O link organized as an independent node (see Figure 2.2b). These nodes are then linked with a fast *interconnect*, which can be PCIe itself or a proprietary interconnect (e.g. Intel UPI).

This independent node architecture is often called *non-uniform memory access* (NUMA), but its implications extend beyond memory access, as we will demonstrate below. I/O devices such as network cards and storage drives are most of the time furnished with a single PCIe link, and as a result each device is affiliated with a single NUMA node, which we call the *home node* of that device. It follows that any communications (e.g. register writes, interrupts, DMA operations) between a device and a node that is not its home node (or *remote node* for short) would suffer extra overhead, due to them needing to cross the NUMA interconnect between nodes. This situation is called *non-uniform I/O access* (NUIOA) [29]. NUIOA not only affects I/O latency due to the longer signal path, but may also affect the maximum available I/O

bandwidth when the workload is constrained by interconnect bandwidth. Therefore, for I/O intensive workloads to achieve optimal performance under NUIOA architectures, the tasks performing I/O (in both kernel-mode and user-mode) need to be located on the corresponding device's home node (see Figure 2.2b).

Locality of devices and virtualization. With ever-increasing performance of I/O devices and system buses such as PCI Express, servers are often equipped with only a single device of each I/O type. In other words, each server would have one network adapter, one NVMe storage device, and so on. Thus, I/O application performance both inside and outside VMs depend on the locality of the application in regards to the device it interacts with. By default, hypervisors do not relocate applications depending on their NUIOA affinity, leading to wasted performance in case of applications that perform direct device I/O (e.g. userspace drivers, RDMA applications, etc.) Moreover, some current hypervisors like Xen do not expose device locality to guest OSes, therefore causing VMs to be unaware of this association and unable to make the appropriate scheduling decisions.

NUMA in virtualization. Hypervisors often use one of two approaches for handling VM resources on NUMA architectures: memory interleaving or vNUMA. Memory interleaving is the default allocation strategy on hypervisor such as Xen [10]. It consists of allocating the VM memory by regions of 1 GB with a round-robin algorithm on each NUMA node, then presenting a Uniform Memory Architecture (UMA) to the VM. vNUMA involves presenting to the VM a virtual NUMA topology which maps its resources to virtual NUMA nodes. It is supported by modern hypervisors such as Xen, VMware and Hyper-V. vNUMA can take advantage of existing operating systems' NUMA awareness to improve VM guests' CPU and memory allocation locality.

2.4 The NVM Express specification

The Non-Volatile Memory Express (NVMe) specification [30] has been widely adopted as a way to remedy I/O inefficiencies between the storage device and operating system. At the core of NVMe is the concept of I/O queues, where multiple independent storage operations can be performed simultaneously without the cost of synchronization. This highly-optimized design has massively benefited storage device and application scalability. Indeed, NVMe devices have managed to reach impressive performance figures; for example, the Intel Optane P5800X series claims a performance of up to 5 million I/O per second and a 99th percentile latency of less than 6 μs [31].

To summarize, the NVMe specification defines a communication protocol between software (the "host") and storage devices (the "controllers"). It specifies an *admin command set* for the host to interrogate and manipulate the controller, and various other command sets for each individual use case: the *NVM command set* for traditional block devices; and the *NVMe-KV command set* for devices having a key-value interface instead of the traditional block device interface.

NVMe provides a generalized *command queue* abstraction regardless of command set. The host sends I/O commands to a controller via *submission queues* (SQs); the controller processes each command and puts its result into a corresponding *completion queue* (CQ). Aside from a dedicated SQ/CQ pair for admin commands, each NVMe controller can communicate with the host using up to 65535 queues, and each queue is further capable of containing up to 65535 commands being processed in parallel. Each queue is a lockless producer-consumer ring buffer; as such, each CPU can communicate with the controller using a dedicated queue, removing the need for synchronization between CPUs when submitting requests. In addition, NVMe allows a N-to-1 correlation between SQs and CQs; in other words, multiple SQs can be associated to the same CQ. The host can wait for completion notifications from a controller in two ways: it can either receive interrupts from the controller, or continuously poll its CQs for any new entries (called *busy polling*, a.k.a. *active polling*).

The NVMe specification allows the use of various transports over which I/O data can flow, such as a *memory transport* for devices attached to a system bus like PCIe, *message transport* over TCP or Fibre Channel, or a *RDMA-based transport* for high-speed remote storage over Infiniband or converged Ethernet. NVMe's support for multiple transports lets operating systems and applications use the same driver and software stack regardless of the underlying connection.

In summary, NVMe's scalable protocol and feature set enables countless new use cases: remote storage, intelligent tiering, key-value databases, etc. NVMe enjoys widespread support from numerous hardware and software vendors, and is poised to become a prominent all-purpose storage protocol.

2.5 Linux's eBPF engine

Berkeley Packet Filters (BPF) [32] was introduced in the BSD operating system for packet inspection, filtering and capturing. BPF makes use of *BPF filters* written in a virtual machine-based language that lets one filter program process multiple protocols at different network layers. Linux originally adopted BPF for the same purpose in its socket filter [33].

The BPF instruction set was extended in Linux into *Extended BPF* (eBPF) with extra instructions and registers. Before running each eBPF program, the Linux kernel verifies its safety through a large range of properties, including constraints on memory accesses, loops and program size. eBPF programs can call a list of authorized kernel helper functions; however, this approach requires recompiling and reinitializing the eBPF verifier every time a new helper function is needed. Linux eBPF is currently employed in various use cases, such as system call filtering (via the Seccomp-BPF API), kernel tracing, LSM security controls, or infrared signal decoding. Notably, its Express Data Path (XDP) feature executes eBPF programs at the earliest points of network packet reception, such as in the network driver or directly inside SmartNIC hardware for the purposes of packet classification and routing [34].

Chapter 3

HyperTP: A unified approach for live hypervisor replacement

Contents

3.1 Overview **16**

3.2 Design of HyperTP **18**

3.2.1 Principles 18

3.2.2 In-place hypervisor transplant 20

3.2.3 Migration-based hypervisor transplant 21

3.2.4 Using hypervisor transplant in a datacenter 21

3.3 Prototype **23**

3.3.1 UISR and device management 24

3.3.2 Implementing InPlaceTP 25

3.3.3 Implementing MigrationTP Xen-to-KVM transplantation . . . 28

3.4 Evaluation **29**

3.4.1 Experimental setup 29

3.4.2 Time breakdown 30

3.4.3 Impact on applications 34

3.4.4 Hypervisor update 38

3.4.5 Hypervisor security 39

3.4.6 Memory overhead 39

3.5 Discussion **40**

3.6 Related works **41**

3.6.1 Hypervisor update 41

3.6.2 Hypervisor security 42

3.7 Summary **43**

3.1 Overview

Chapter 1 showed that the increasing usage of virtualization in modern datacenters is accompanied with a simultaneous increase in the need for regular preventative maintenance and updating of hypervisors and related software. Hypervisor updates are often done for one of two reasons: either for introducing new features, or to mitigate a certain vulnerability. These updates involve one of several methods: (1) a full reboot of the host and all running guests; (2) live migration of running VMs from a host running old versions of the hypervisor to another host running updated software; and (3) live patching of the running hypervisor. Each method has its own set of limitations: full reboots are highly disruptive to the infrastructure's operations; live migration consumes large amounts of time and network bandwidth, and prevents the usage of certain virtualization features; live patching is limited to small fixes, mostly of security issues, and requires extra development effort to create a customized livepatch for each fix, as automated patch generation does not guarantee that a patch is safe to apply [35, 36]. Therefore, there remains a need for a timely and efficient hypervisor maintenance system capable of delivering both feature and security updates without causing service disruption or restricting desirable features.

To address this need, we introduce the concept of *hypervisor transplant*. Our goal is to quickly replace one running hypervisor H_{current} with another H_{target} *without rebooting running VMs* for the purpose of speeding up preventative maintenance of virtualization infrastructure. Note that H_{target} can be anything from an updated version of H_{current} to a completely different hypervisor, allowing more flexibility in choosing an appropriate hypervisor for the required workload.

We materialized our concept of hypervisor transplant in a platform called *HyperTP*, which combines in a unified way two complementary approaches: *in-place micro-reboot-based transplant* (noted InPlaceTP) which replaces a running hypervisor with little downtime and no extra resources, and *live VM migration-based transplant* (noted MigrationTP) which causes almost no VM downtime. The combination of these two approaches answers the constraints put on both applications running in VMs and on datacenter infrastructures.

Indeed, InPlaceTP and MigrationTP present a tradeoff between maintenance deadline, downtime tolerance and upgrade resource availability. For instance, InPlaceTP's micro-reboot-based transplant requires several seconds of downtime. However, such downtime figures are not without precedent. Namely, Microsoft Azure presents downtimes of up to 30 seconds for maintenance operations [37]. Orthus [9] reports figures of up to 9.8 seconds for VMM upgrades. Similarly, Hy-FiX [38] requires 8.1 to 12.3 seconds of downtime for the same task. In exchange for an extended downtime, InPlaceTP, by its in-place nature, does not require large amounts of extra resources and significantly shortens the maintenance timeframe. In comparison, MigrationTP causes minimal downtime to running VMs but with the additional cost of spare machines and network bandwidth, like other live-migration-based maintenance operations [9]. In the current state of HyperTP, it is up to the datacenter operator to decide which transplant approach is the most appropriate for their maintenance operation, since equivalent

policies are already provided for dealing with periodic platform updates.

While live migration and micro-reboot are known approaches, the main novelty in designing HyperTP is to ease the support of *multiple different hypervisors*. Naturally, this raises the question of managing the heterogeneity of their VM state representations. To resolve this, we build both approaches of HyperTP around two common principles, a *VM state hierarchy* which identifies and defines the various types of memory states in relation to their functionalities relative to a VM's operation, and an *Unified Intermediate State Representation* (UISR) to facilitate the creation of HyperTP-compatible hypervisors.

We demonstrated our platform by re-engineering Xen and KVM, the two most popular open source hypervisors, into HyperTP-compliant hypervisors. They represent the two types of hypervisors: type-1 (Xen) and type-2 (KVM), thus demonstrating the scope and flexibility of our solution. We evaluated our prototype at a machine scale to validate its ability to transplant both idle VMs as well as active VMs running various types of benchmarks. We also presented the downtime incurred by HyperTP while running various workloads such as SPEC CPU 2017, MySQL and Redis.

We investigated the usage of HyperTP at a cluster scale in a datacenter. We highlighted two direct usages of the platform: for *hypervisor updating*, where HyperTP shortens the time and reduces the resources needed to apply a new hypervisor version; and for *hypervisor security*, where HyperTP helps reduce the time window where a virtualized infrastructure is exposed to known vulnerabilities.

To summarize, in this chapter, we present the following contributions:

- We present HyperTP, a two-pronged solution including MigrationTP and InPlaceTP to help simplify hypervisor updates and maintain hypervisor security.
- We implement HyperTP in multiple directions: Xen→KVM, KVM→Xen, and Xen→Xen, thus demonstrating HyperTP's scope and flexibility.
- InPlaceTP Xen→KVM causes minimal downtime to running VMs (1.91 seconds for a VM with 1 vCPU and 1 GB of RAM), with negligible memory and I/O overhead and without requiring VM reboots. With KVM→Xen and Xen→Xen, the downtime is about 7.8 seconds for the same VM configuration. MigrationTP offers similar performance to traditional homogeneous VM live migration.
- We show the benefits of InPlaceTP over migration-based solutions for upgrading an existing virtualization cluster. Namely, we demonstrate that upgrading 10 servers each running 10 VMs using InPlaceTP for 80% of the VMs takes 3 minutes and 54 seconds while using MigrationTP alone would take up to 19 minutes.
- We conduct a study of vulnerabilities in Xen and KVM over the last 7 years. We observe that most vulnerabilities are specific to a single hypervisor and caused by faulty implementations, and show how HyperTP can be used to reduce vulnerability windows of virtualized infrastructures.

The rest of the chapter is organized as follows. Section 3.2 present the general overview of HyperTP. Section 3.3 presents the implementation of HyperTP. Section 3.4

presents the evaluation results, followed by Section 3.5 which discusses the limitations of our approach. Section 3.6 discusses the related works. Finally, Section 3.7 concludes this chapter.

3.2 Design of HyperTP

In this section, we first present the two main principles of the design of HyperTP, *VM state hierarchy* and *Unified Intermediate State Representation*. We then show how these principles are applied to InPlaceTP and MigrationTP, and demonstrate their application in our aforementioned use cases.

3.2.1 Principles

To reiterate, the main goal of HyperTP is to rehost a VM running on one hypervisor to another hypervisor without causing a VM reboot. Let us note H_{current} and H_{target} as the current and target hypervisors of the transplant process respectively. A hypervisor transplant is conducted by performing the following five generic work items:

1. Suspend running VMs;
2. Translate VM states into the UISR neutral format;
3. Transfer VM states to the new target hypervisor;
4. Restore VM states from UISR to H_{target} format;
5. Resume VMs and finish the transplant operation.

Note that the aforementioned workflow is not meant to be taken in strict sequence; we optimize the contents and ordering of this workflow depending on the scenario being executed (InPlaceTP or MigrationTP). These optimizations are described in the next sections of this chapter.

VM state hierarchy. Generally, we consider that the VMs' states include all the data structures in the hypervisor for the management of virtual resources (CPUs, memory, devices). Following Section 2.2, we observe that a VM's in-memory representation consists of multiple types of data, where each data type needs to be translated in a different way. For example, a guest memory page needs to be treated differently from a scheduling object associated to a vCPU. Nevertheless, different hypervisors running on the same platform typically aim to provide a common-ground virtual hardware that accomodates most guest OSes; not to mention, these hypervisors necessarily share some common behaviors by virtue of running on the same architecture. This implies *there exists a commonality between how different hypervisors manage their internal states*.

In HyperTP, we propose a hierarchy of memory resources in a VM, which serves to inform us of which kinds of data need to be kept as-is, transformed or discarded. Our resource hierarchy is divided into four main categories:

- *Guest State*, like its name, represents the memory states that are specific and visible to the VM, like memory pages. During transplantation, guest states require the least transformation, i.e. they can stay mostly untouched throughout the whole process.

- *VM_i State* corresponds to data structures that are specific to the execution of one VM, but are not necessarily visible to the VM in their raw form. An example of *VM_i State* are 2D page tables (2DPT) or vCPU register states. In fact, while the structure and content of the 2DPT is usually specific to the hardware virtualization technology being used (e.g. Intel's EPT, AMD's NPT), each hypervisor has its own policies for managing the 2DPT, and therefore the contents of each VM's 2DPT are not directly translatable between hypervisors. Similarly, while vCPU register states are closely linked to the vCPU's execution, each hypervisor saves a vCPU's states in its own different data structure, and therefore these states must be translated if a VM were to be transplanted between different hypervisors.

- *VM Management States* are in-memory states that serve to manage the VM, but do not necessarily contain the VM's state itself. For example, a hypervisor's scheduler queue might refer to a VM's vCPUs, but does not contain any vCPU states. In general, these states can be easily reconstructed if necessary from the previously-mentioned types of state.

- *HV State* finally represents the set of hypervisor states that are not specific to any VM, such as the memory assigned to hardware drivers. HyperTP does not save or transform these states; they are considered to be disposable (in the case of InPlaceTP, where they are reinitialized using micro-reboot) and/or reconstructable (in the case of MigrationTP, where the migration does not take them into account).

Unified Intermediate State Representation. From the hierarchy of VM states presented above, we observe that many types of VM states are at least partially specific to each hypervisor that is managing the VM. Yet, it is unreasonable to demand that all hypervisors use the same data structures for its functionalities in order to support hypervisor transplant, since such standardization not only limits the range of functionalities each hypervisor can potentially support, but also might introduce accidental common vulnerabilities.

To realize the concept of hypervisor transplant, we transform each VM's states into a UISR. UISR represents the hypervisor-dependent state of each VM with a hypervisor-independent intermediate representation that facilitates the transfer of VM states across different hypervisors. In this sense, UISR shares the same objectives as the network-level neutral data representation XDR [39]. Relying on a neutral format simplifies the re-engineering of a hypervisors into a HyperTP-compliant one, since the hypervisor developer only has to understand the UISR format instead of the representation formats of all existing hypervisors.

The goal of UISR is to sufficiently represent a VM for its reconstruction in a HyperTP-capable, compatible hypervisor. Following the VM state hierarchy presented above, we posit that *knowledge of* Guest State *and* VM_i State *is sufficient for the reconstruction of a VM*. These states are therefore the targets of our UISR. In general, the following VM states are collected by HyperTP and distilled into UISR: VM memory pages; CPU registers and control registers; interrupt controller and timer states; and virtual hardware states (including *hidden states* required to reconstitute the virtual device). However, we acknowledge that the above list is not an exhaustive list of all VM states; the restorability of VMs under HyperTP depends on the compatibility of their configuration and the UISR format.[3]

To transform a VM's states between its hypervisor-specific representation and UISR, each hypervisor needs to implement a pair of translation functions for each class of VM state. These functions can be as simple as the identity function for hypervisors that directly use UISR as their internal VM states, or an explicit translation from a hypervisor's internal state to the corresponding UISR. Nevertheless, knowing that hypervisors often share certain commonalities (as argued in the above section), we expect most hypervisors to be able to support UISR without needing extensive modifications.

3.2.2 In-place hypervisor transplant

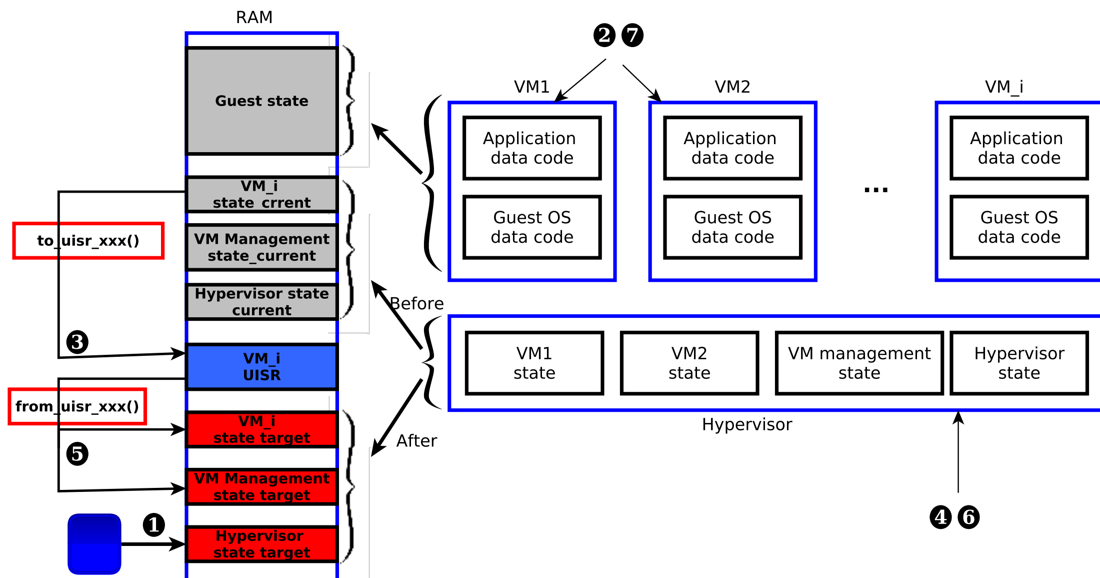

Figure 3.1: Basic workflow of InPlaceTP.

Figure 3.1 summarizes the working principles of InPlaceTP. In short, InPlaceTP performs a hypervisor transplant through the means of replacing the running hypervisor. To accomplish this, step ❶ first loads the target hypervisor into memory. After pausing any running guest VMs to be transplanted ❷, we invoke the corresponding UISR

[3]Note that this limitation is present in same-hypervisor live VM migration as well; current hypervisors prevent VMs using certain hardware features from being migrated.

translation functions to convert the corresponding *Guest/VM$_i$ States* to the UISR format ❸. We perform a micro-reboot to hand over control of the hardware to H_{target} ❹, while passing to it any relevant UISR *Guest/VM$_i$ States*. H_{target} then converts the received *Guest/VM$_i$ States* into its own native format ❺, and uses these states to reconstruct the VMs ❻. Finally, the VMs to be transplanted are resumed ❼ and the transplantation process is completed.

For the purpose of HyperTP, *Guest States* refer specifically to memory pages owned by the VM and used as its memory. These pages naturally do not require any specific transformation or rewriting to be converted into UISR. As long as these pages remain intact, they can be easily reincorporated into the new VM. That is why during the entire process, InPlaceTP ensures that these *Guest States* are protected from accidental deletion and corruption. The steps ❸ and ❺ in fact simply involve recording their location in the host's physical memory, and giving them back to the VM afterwards. This represents a large time saving for InPlaceTP as costly memory copies and disk writes are minimized.

3.2.3 Migration-based hypervisor transplant

Under MigrationTP, the target of transplantation is no longer the current server, but rather a remote server, in the same way as normal VM live migration. As such, MigrationTP follows the same procedure as VM live migration, which largely matches up with HyperTP's own steps. The differences come during the sending of VM state to the destination server; during this step, MigrationTP makes use of *state proxies* to translate the VM's *VM$_i$ States* into UISR. Note that these proxies are based on the same state transformations as used by InPlaceTP. On the destination server, another proxy then translates the UISR back into H_{target}'s VM state format. While *Guest States* need to be copied over network to the destination server unlike InPlaceTP, this copying step does not need to involve the state proxy.

3.2.4 Using hypervisor transplant in a datacenter

In summary, HyperTP is a combination of two related approaches InPlaceTP and MigrationTP, based on a common UISR for the representation of VM states. In this section, we elaborate on the application of HyperTP to two exemplary use cases: firstly, for installing updates on virtualized infrastructure; and secondly, for the mitigation of hypervisor vulnerabilities.

Hypervisor update. As stated in Section 3.1, hypervisor updates are used not only to fix system bugs but also to deploy new software features. A public cloud provider might wish to quickly upgrade their hypervisor fleet to add new services and to increase feature velocity of their cloud solution; a private cloud customer might instead want to install a new hypervisor version to maintain software support, or to fix performance and reliability issues, etc. However, existing hypervisor update methods are either disruptive or limited in scope, causing risk-averse operators to hesitate applying updates.

In comparison, HyperTP offers datacenter operators rapid deployment of new software with InPlaceTP in complement with other upgrade choices (MigrationTP, normal migration or live patching). Additionally, InPlaceTP is not only capable of upgrading the current hypervisor core, but can even completely replace it with a different hypervisor altogether. This is especially useful e.g. in cases where the operator wishes to switch their hypervisor vendor, where the two different hypervisors can be managed using the same tooling (like OpenStack). In conclusion, HyperTP helps facilitate the deployment of desirable updates while minimizing their impact on the infrastructure.

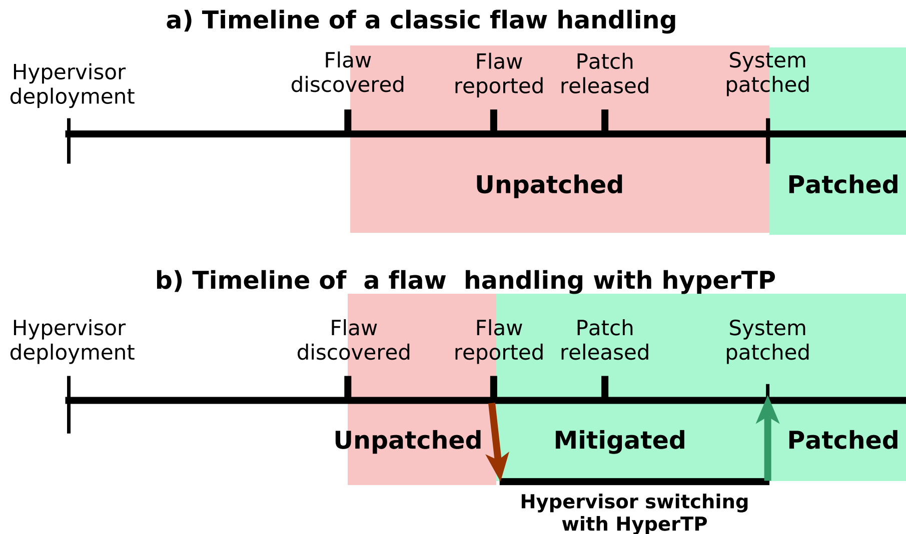

Figure 3.2: (a) Traditional vulnerability mitigation in data centers and (b) our hypervisor transplantation-based solution.

Hypervisor security. Hypervisors are continuously subject to multiple security vulnerabilities. Similar to previous works [40], we define the *hypervisor vulnerability window* regarding a given security flaw as the time between the identification of said flaw (whether by a good or bad actor) and the integration of a patch in the running hypervisor (see the red zone in Figure 3.2a). In fact, the vulnerability window is the sum of two durations: (1) the time required to propose a patch once the vulnerability is discovered; and (2) the time to apply this patch in the system. The time to release of a patch is highly dependent on the corresponding vulnerability's severity, and can vary from one week with vulnerabilities such as the MD5 collision attack [41], to 7 months with vulnerabilities such as Spectre and Meltdown [42, 43].[4] Meanwhile, the time to apply a patch mainly depends on the datacenter operators' patching policies. Together, this timeframe leaves plenty of time to launch an attack against a vulnerable installation.

To alleviate this issue, HyperTP can be used to preemptively and temporarily replace the actual datacenter hypervisor (e.g. Xen) with a different hypervisor (e.g. KVM) which is immune to the given vulnerability (see Figure 3.2b). This approach mitigates

[4]Note that Spectre and Meltdown are CPU-specific vulnerabilities with CVEs declared on Intel products. Hypervisors and operating systems were not directly concerned by the CVE declaration.

the impact of a vulnerability given one of the following conditions is met: (1) there is already a known-safe hypervisor when the vulnerability is discovered, and (2) a patch solving the vulnerability can be developed in a shorter amount of time for an alternate hypervisor than the one used in the datacenter.

Table 3.1: Number of Xen and KVM critical and medium vulnerabilities per year.

Year	Xen	KVM	Common
2013	3	3	0
2014	4	1	0
2015	11	1	1
2016	6	3	0
2017	17	1	0
2018	7	2	0
2019	7	2	0
Total	55	13	1

To investigate the viability of hypervisor transplant in this context, we collected a list of critical vulnerabilities over the last 7 years for Xen and KVM (see Table 3.1). A vulnerability is considered as critical when its Common Vulnerability Scoring System 2.0 score is higher than 7 [44]. Over that period, we found only one common critical vulnerability (CVE-2015-3456) originating from QEMU, a common component used by both Xen and KVM. This low number supports our starting assumption that a safe alternate hypervisor exists. Overall, the number of critical vulnerabilities per year remains low, which means that even if hypervisor transplant cannot be done too frequently, it would still bring an improvement in security.

3.3 Prototype

We implemented HyperTP on top of two commonly-used open-source hypervisors, Xen and Linux KVM. We used Xen 4.12.1 with fully-virtualized HVM domains; Xen PV was not used due to its tight coupling with the Xen API, which makes moving PV-based VMs away from Xen more difficult than moving HVM-based VMs. On the KVM side, we used Linux 5.3.1 along with the standalone kvmtool. We implemented InPlaceTP in three transplantation directions: Xen→KVM and KVM→Xen as examples of our heterogeneous hypervisor transplant; and finally Xen→Xen as an example for enabling live upgrade of Xen hypervisor instances. KVM→KVM InPlaceTP can be implemented with the same principles; however, we did not implement this scenario as it is already covered by existing works [9, 38]. MigrationTP was additionally implemented for the Xen→KVM direction. We configured our VMs to use remote storage to concentrate I/O activity onto virtual networking.

Our HyperTP prototype represents a total of approximately 8.5 KLOC, of which 2.2 KLOC belong to the hypervisors, 5.2 KLOC in userspace management tools (*libxl*, kvmtool and Kexec), and 1.1 KLOC for HyperTP orchestration purposes. We based our prototype mostly in userspace; in fact, only 10% of our implementation involves

Xen/Linux kernel code. Such a prototype takes advantage of existing tools and libraries for controlling VMs (*libxenctrl*, kvmtool), therefore being highly compatible with different versions of Xen and Linux/KVM. Our implementation of HyperTP also runs the bulk of its code with minimal privilege and only during the transplant process, thus minimizing HyperTP's security footprint.

In the following sections, we describe the common implementations of VM state management with UISR, as well as our implementations of InPlaceTP and MigrationTP in detail.

3.3.1 UISR and device management

When considering only HyperTP from Xen to Xen itself, our translation and restoration functions can simply be the identity function. However, when transplanting VMs between Xen and KVM, the need arises for a common UISR format that can be used to represent VM states. Xen provides a relatively stable and well-defined VM state format that covers the majority of VM-critical hardware. As a result, we use a slightly modified and extended version of Xen's VM format as our UISR.

Platform device management. To recall Section 2.2, we use the term *platform device* to refer to non-replaceable devices inside the VM that are critical to its execution (CPU registers, interrupt controllers, timers), in contrast to *bulk I/O devices* such as network and storage devices that the guest OS can function without. Naturally, platform devices attract special attention during the hypervisor transplant process, since they are needed for the guest OS to safely resume its operation.

We first established a mapping to be implemented in our state translation functions between each of a VM's platform components on Xen and KVM and our UISR equivalent. Subsequently, we extracted the native VM states of Xen and KVM using the appropriate system calls (using *libxenctrl* on Xen and IOCTLs on KVM). Our translation functions then handled the reading and writing of various VM formats, and at the same time provided fixes to certain platform components that are not mutually compatible between the two hypervisors. Table 3.2 shows the correspondence between UISR states and native VM states of Xen and KVM in our implementation.

Table 3.2: Correspondence between hypervisor platform device states and UISR on x86.

Xen HVM	**UISR**	**KVM**
CPU regs	CPU	(S)REGS, MSRS, FPU
LAPIC	LAPIC	MSRS
LAPIC regs	LAPIC_REGS	LAPIC_REGS
MTRR	MTRR	MSRS
XSAVE	XSAVE	XCRS, XSAVE
IOAPIC	IOAPIC	IRQCHIP
PIT	PIT	PIT2

Management of bulk I/O devices. Generally speaking, virtual device functionalities are presented to VMs in one of two main types: device passthrough and device emulation. Our handling of I/O devices depends on the type of device, the details of which are presented below.

Device passthrough grants a VM direct access to hardware installed in its physical server. The VM therefore communicates with hardware using the same driver as a native system would. To prevent the VM from abusing passthrough hardware to attack the host, a *I/O memory management unit* (IOMMU) is often used to limit that hardware's access to physical memory. Device passthrough gives VMs an I/O performance that is as close as possible to native performance, but also binds the VM to its underlying hardware, and therefore requires special attention in certain cases (e.g. during live migration). In the case of HyperTP, we treat passthrough-assigned devices the same way as a live migration event: the VM is informed of the transplantation event and performs the necessary steps to stop its device driver in a consistent fashion. Once the transplantation event completes, the device is reconnected to the guest and resumes its operation.

Emulated devices (including paravirtualized ones) are implemented in software through the use of trap-and-emulate techniques. However, HyperTP might lead to a change in the software that is performing the emulation. In our solution, emulated devices can be handled using one of two ways: either by copying and translating the emulation state for use in the target hypervisor (as we've done with our emulated platform devices), or by using the same stop-reconnect technique described above. The stop-reconnect technique has the advantage of being able to replace the paravirtualization API being used by the VM; in other words, a VM making use of Xen's *netfront* driver can later switch to a *virtio* network driver on KVM after a successful hypervisor transplant.

We added a kernel module in the guest that listens for transplant events from the host. The kernel module is responsible for suspending running processes inside the guest and preparing them for the I/O device transition. Notably, this technique does not break existing network connections and therefore does not interfere with applications running on the VM outside of the expected downtime, as the guest's in-kernel connection states are not altered by our device replacement.

VM memory management. The treatment of VM memory differs depending on the approach being used. Regardless, VM memory, as part of the *Guest States* category of VM states, is transferred over to the new hypervisor without needing modifications. We describe the memory handling of each approach in the below sections.

3.3.2 Implementing InPlaceTP

Following the general workflow of HyperTP described in Section 3.2.1, we observe that InPlaceTP requires the customization of steps 2 (translation), 4 (restoration) and especially step 3 (micro-reboot). Besides the VM state transformation described

above, we detail the technical aspects of InPlaceTP's VM memory management and micro-reboot implementation.

VM memory management. VM memory is often made of hundreds to thousands of fragments spanning multiple gigabytes in size. It therefore deserves special attention in InPlaceTP, as memory copies, or worse, disk writes must be minimized during the transplantation process to avoid causing excessive downtime. As pointed out in Section 3.2.2, our solution is to keep a VM's memory in place during the entire transplantation process, while protecting it from accidental corruption while the new hypervisor is reinitialized. To accomplish this, we store a representation of our transplanted VMs in an in-memory filesystem structure called a *PRAM structure*, adapted from the PRAM patchset [45]. Figure 3.3 shows the detailed construction of our PRAM structure, which consists of metadata pages that record the physical location of a VM's memory pages, allowing each VM's memory to be reconstructed after the new hypervisor boots up. In short, the PRAM structure consists of a list of *file pointers*, each of which points to a *file information page* which identifies an individual VM's unique name and memory size. Each file information page then points to a linked list of *page entries*; each page entry maps a range of VM memory pages to its location in physical memory. This memory can later be mapped in the restoration step 4 with an appropriate API (e.g. *mmap*).

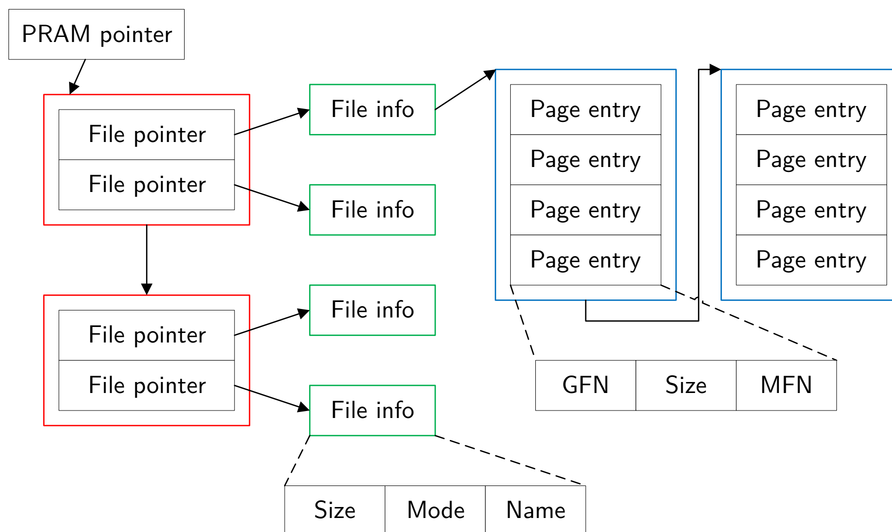

Figure 3.3: PRAM structure, used for identifying VM memory pages.

Micro-reboot. Step 3 in the hypervisor transplant process requires quick and efficient switching between two hypervisors on the same system without corrupting the VM states already stored in host memory. For this purpose, we make use of Kexec, which allows quickly booting a new OS kernel without having to reinitialize all devices.

While booting a new host kernel with Kexec does not require passing through BIOS (and therefore resetting memory contents), the new host kernel can still overwrite any VM states living in host memory when the system is being reinitialized. To protect

the in-memory VM state information from being corrupted during transplantation, we made three main modifications to the Kexec process:

- We reserved a dedicated memory location in which to load the new kernel to avoid Kexec itself from overwriting the relevant PRAM pages.
- We passed the PRAM pointer to the new host kernel during Kexec via its kernel command line. Knowing the PRAM structure's location, we modified the target kernel (Xen and Linux) to protect the PRAM pages and VM memory contents from being accidentally overwritten. This protection is applied during each kernel's early boot process. After the new kernel boots up, each VM's memory is presented again in a virtual filesystem where it can be picked up and reinjected back to its corresponding VM.
- Finally, we implemented various fixes in Kexec and the hypervisor kernels to ensure that the new hypervisors operate correctly after InPlaceTP. This ranges from correctly initializing boot structures provided by the Kexec userspace tools to ensuring that hardware devices keep operating on the new host kernel.

Optimizations. The main limitation of InPlaceTP is that of its required downtime. Particularly, as InPlaceTP suspends the VM when its host kernel is being replaced, any transplantation step happening in the meantime will impact the overall downtime of the system. However, our general hypervisor transplant workflow is not meant to be fixed, and as a result leaves room for certain optimizations. For the interest of reducing downtime during the transplantation process, we implemented the following optimizations in our InPlaceTP prototype:

- We *optimized the execution ordering* between different transplantation steps in the same spirit as the pre-copy step of VM live migration. This optimization is implemented in two aspects. Firstly, we build PRAM substructures ahead of time before the VM ever gets suspended. Especially in the case of Xen, where VMs' memory pages are preallocated ahead of time, this step presents a significant time savings as very little work remains once the VM is finally suspended. Secondly, as a small optimization, we load the new hypervisor kernel well before the final Kexec boot. In short, the step of suspending all running VMs is deferred until absolutely necessary, thus helping to reduce InPlaceTP's downtime.
- We *parallelized* our VM state construction process in order to significantly speed up the translation of VM_i *States* and creation of PRAM structures needed for transplanting VMs.
- We implemented *large page support* natively inside InPlaceTP. The large page CPU feature, commonly used by hypervisors, allows combining multiple appropriately-aligned pages into one large page to lower the overhead of 2D page translations. For example, on the x86-64 architecture, 512 consecutive 4 KB pages can be combined into a 2 MB large page. With our optimization, we encode each large

page's *order* (i.e. level of page combination) in our PRAM structure; each large page therefore takes up only one page entry (see Figure 3.3). Not only does this speed up our PRAM construction by lowering the number of VM pages that must be traversed, it also reduces the overhead of PRAM structures w.r.t. VM memory size.

- We adapted Linux/KVM to *prioritize the VM restoration process*. We found that on our test platforms, the default service priorities necessitated a long wait before VM restoration could begin (as reported by the *systemd-analyze* tool). In response, we adjusted the service boot priority on our host operating system to resume all VMs as soon as the basic services required by the VMs are ready. On a busy host running multiple services, this optimization again serves to minimize VM downtime, in lieu of waiting for non-critical, unrelated services.

3.3.3 Implementing MigrationTP Xen-to-KVM transplantation

As stated in Section 3.2.3, the workflow of MigrationTP closely matches that of InPlaceTP, as well as that of a normal live VM migration. Based on the existing InPlaceTP device management primitives, we implemented the necessary VM state transformations on the destination kvmtool. The incoming migration state stream from Xen is translated accordingly and applied to the destination VM. We describe below each individual step of MigrationTP Xen→KVM in detail:

- *Pre-copy:* In this step, VM memory pages are copied from the source host to the destination host over several iterations while the VM continues its execution. Page modification tracking is used to keep track of which pages have changed since the last copy iteration. Similar to InPlaceTP, the received memory pages do not need additional transformation and MigrationTP applies them as-is.

- *VM suspension:* Once a sufficient number of VM memory pages has been transferred to the destination, the source VM is suspended to prepare for the final migration step (cf. workflow step 1).

- *State transfer and reconstitution:* This phase covers steps 2 to 4 of the general HyperTP workflow. In this step, the individual device state translations described above are applied to the incoming migration stream containing Xen VM states to form our UISR; kvmtool then receives the resulting UISR and applies it to the destination VM using the appropriate KVM API calls.

- *Starting the destination VM:* We arrive at the final step 5 of our HyperTP workflow. Once all necessary VM states have been transferred and successfully applied, following positive confirmation from both the source and destination hypervisors, the source VM is destroyed and MigrationTP puts the destination VM back into operation.

3.4 Evaluation

This section presents the performance evaluations of the two approaches of HyperTP, InPlaceTP and MigrationTP. In particular, we present a time breakdown of each approach under various configurations, as well as their impacts on several different kinds of application workloads. We also evaluate the impact of HyperTP on the two use cases of hypervisor update and hypervisor security, and its various memory overheads. Our evaluations aim to answer the following questions:

- What are the time and memory costs incurred by each step of the transplantation for both approaches, InPlaceTP and MigrationTP?
- How scalable is each approach with varying VM sizes and number of VMs?
- What is the performance impact of HyperTP on user applications?
- How does HyperTP perform at the cluster scale for hypervisor update?
- Finally, how does HyperTP help improve security in a datacenter?

3.4.1 Experimental setup

Hardware. For our evaluations, we used two kinds of machines: two machines, each equipped with an Intel i5-8400H CPU and 16 GB of RAM (called *M1*) and one equipped with 2x Intel E5-2650L v4 and 64 GB of RAM (called *M2*). All machines are linked with a 1 Gbps Ethernet connection. We evaluated InPlaceTP on both M1 and M2; to ensure that MigrationTP experiments were conducted between similar machines, we performed them on M1 machines only. We reserved 2 CPUs for the administration OS (dom0 in Xen and host Linux in KVM), and configured hypervisors to use 2 MB huge pages for guest memory. In our cluster-scale evaluations, each cluster member was equipped with 2x Intel Xeon E5-2630 v3 and 96 GB of RAM. All machines in the cluster were connected together with 10 Gbps Ethernet.

Applications. We evaluated HyperTP using three main application types: SPEC CPU 2017 (CPU-heavy), MySQL (database) and Redis (memory-heavy). Table 3.3 shows a list of our tested workloads in detail.

Table 3.3: Description of HyperTP evaluation workloads.

Benchmark (metric)	Description
SPECrate 2017 (execution time)	23 CPU- and memory-intensive workloads
Sysbench MySQL 5.7 (latency)	Stressing a relational database with a SQL load injector
redis-benchmark (QPS)	Stressing an in-memory KV store with its included load injector

3.4.2 Time breakdown

In this experiment, we aim to analyze the duration of each phase of InPlaceTP and MigrationTP for each transplantation direction. We used idle VMs for this evaluation since VM activity does not impact the transplantation time.

For InPlaceTP, we break down the transplant process into four steps: (1) PRAM structure construction, where the VM's memory layout is analyzed and stored into a PRAM structure (noted as *PRAM* in Figure 3.4); (2) UISR translation, where the VM is suspended and then its execution state is taken and translated into UISR (noted as *Translation*); (3) micro-reboot, where the target hypervisor and supporting software are started using Kexec (noted as *Reboot*); and (4) UISR restoration, where the previously-taken UISR is used to restore and consequently resume the VM (noted as *Restoration*). Since our PRAM structure is constructed before pausing VMs, the downtime therefore equals *Translation* + *Reboot* + *Restoration*. Note that $time = 0$ on Figure 3.4 corresponds to the moment when VMs are paused, therefore the *PRAM* step is always located below the x-axis. Since network services are not needed for all application types, we present its initialization time separately from the overall transplant time (noted *Network*). Therefore, this time will not be counted in the downtime of network-independent applications, such as the SPEC CPU2017 benchmark, but counted for network-dependent applications.

For MigrationTP, we show the duration that the VM is paused (a.k.a. downtime) and the total migration time. This is in comparison to normal live VM migration, which follows a very similar procedure (without our MigrationTP proxy in particular).

Basic evaluations

This scenario allows us to gather basic information about the performance of each step of our HyperTP workflow. In this scenario, our machines ran a single VM configured with 1 GB of memory and 1 vCPU. This VM size is representative of cloud workloads such as Microsoft Azure [46]. Our smallest machine (M1) could host up to 12 VMs of this size each. We repeated each experiment 5 times, while presenting average values when standard deviation is very low, and box plots otherwise.

InPlaceTP: Xen→KVM (Figure 3.4). We start with a focus on Xen→KVM to detail the time costs of each step of hypervisor transplant. The total transplantation time is 2.03 and 4.74 seconds on M1 and M2 respectively, of which 0.13/0.20 seconds is spent on *PRAM*; 0.05/0.19s on *Translation*; 1.71/3.60s on *Reboot*; and 0.15/0.75s on *Restoration*. *Reboot* is the dominant step of the process, representing 83% and 76% of the total transplantation time on M1 and M2 respectively. The resulting total downtime is 1.91s on M1 and 4.54s on M2. When networking is taken into account, the process takes 6.7s on M1 (of which 6.6s is spent waiting for the network card) and 5.8s on M2 (with 5.6s spent on networking). Despite the long network downtime, we observe that these interruptions do not affect the operation of network connections.

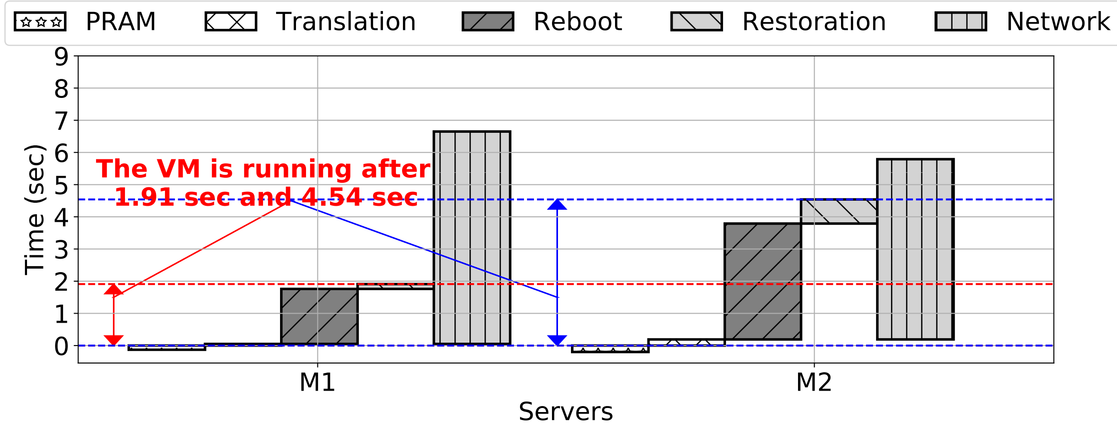

Figure 3.4: Time breakdown of each step of InPlaceTP Xen→KVM with a single VM. See Section 3.4.2 for a description of individual steps and our measuring process.

InPlaceTP: KVM→Xen and Xen→Xen (Table 3.4). We compare different InPlaceTP directions in this experiment. We observe that the reboot times for KVM→Xen and Xen→Xen are higher than that of Xen→KVM:

- KVM→Xen vs Xen→KVM: 6.67s vs 1.71s on M1; 17.92s vs 3.60s on M2;
- Xen→Xen vs Xen→KVM: 6.61s vs 1.71s on M1; 17.84s vs 3.60s on M2.

These differences are mainly caused by Xen's boot process. In fact, being a type-1 hypervisor, Xen requires launching two kernels: the Xen hypervisor and the Linux dom0 kernel. Regardless, we note that the downtime caused by these directions of InPlaceTP is still far from the 30s maintenance window proposed by Microsoft [37] even with several VMs, as we will demonstrate in the next section.

Table 3.4: InPlaceTP for KVM→Xen and Xen→Xen. Downtime in seconds.

	PRAM	Translation	Reboot	Restore	Downtime
M1					
Xen→KVM	0.13	0.05	1.71	0.15	1.91
KVM→Xen	0.22	<0.01	6.67	0.72	7.39
Xen→Xen	0.73	0.06	6.61	0.60	7.27
M2					
Xen→KVM	0.20	0.19	3.60	0.75	4.54
KVM→Xen	0.22	<0.01	17.92	1.23	19.15
Xen→Xen	2.87	0.22	17.84	1.19	19.25

MigrationTP: Xen→KVM (Table 3.5). We demonstrated live migration between two Xen hosts to establish a baseline for analyzing the performance of MigrationTP. Firstly, we observe that the total migration time is almost the same, about 9.5 seconds (dominated by memory page copies). Secondly, the downtime of MigrationTP is 27× lower than that of live migration between two Xen hosts. The reason is that on the destination host, MigrationTP uses kvmtool which is more lightweight compared to Xen's *libxenctrl* and therefore needs less time to resume the VM.

Table 3.5: MigrationTP Xen→KVM compared to Xen VM live migration.

	Xen→Xen	MigrationTP (Xen→KVM)
Downtime	133.59 ms	4.96 ms
Migration time	9.56 s	9.63 s

Horizontal and vertical VM scalability

We evaluate all HyperTP directions (Xen→KVM, KVM→Xen, as well as Xen→Xen) while varying the VM size (number of vCPUs and memory size) and number of VMs running on each machine. Figure 3.5 presents our results on both M1 and M2. Each row of figures corresponds to a transplantation direction (e.g the first row is Xen→KVM on M1 and M2), while each column contains results when varying an experimental parameter (e.g. the first column contain results with varying number of vCPUs on M1).

InPlaceTP scalability. From the first and fourth columns of Figure 3.5, we first notice that the number of vCPUs has no impact on the transplantation time, regardless of the transplantation direction. However, the second and fifth columns demonstrate a slight growth in downtime when varying the VM memory size on both M1 and M2 for all transplantation directions. This is mostly due to the restoration step taking more time with increasing memory size for the VM. Similarly, the total transplantation time increased slightly with the number of VMs, especially in the case of KVM→Xen and Xen→Xen which necessitates the use of Xen's slightly slower VM stack (columns 3 and 6).

Similar to our previous evaluations, we observe that the reboot times of KVM→ Xen and Xen→Xen (second and third rows) are higher than that of Xen→KVM (first row) due to Xen's longer boot sequence. Finally, we can observe that the PRAM time increases in respect to the number of VMs or memory size when the transplantation starts from Xen (first and third rows). This is due to the need to use Xen hypercalls to get access to the memory mapping of VMs for building PRAM structures. Nevertheless, this does not impact running applications because most of the PRAM structure is built with the VMs still running.

In summary, thanks to the fact that we build PRAM before pausing VMs, the VM downtime remains minimal, within 1.91 seconds and ≈10 seconds for M1; and 4.54 and ≈22 seconds for M2. Notably, our results are comparable to that of Orthus [9] (from 0.48 seconds up to 9.8 seconds), which only upgrades the KVM module and QEMU without rebooting the physical machine. Moreover, this downtime is still smaller than the 30s proposed by Microsoft [37] during maintenance windows.

MigrationTP scalability: Xen→KVM. Figure 3.6 presents our results of MigrationTP downtime compared to that of normal VM migration. Generally, MigrationTP downtime is lower than that of Xen→Xen migration because of kvmtool's more efficient stop-and-copy step. Additionally, while this downtime increases slightly with increasing numbers of vCPUs, it is impacted only minimally by the VM's memory size. We use box plots in the last subfigure because of the high variation in downtime induced by Xen when

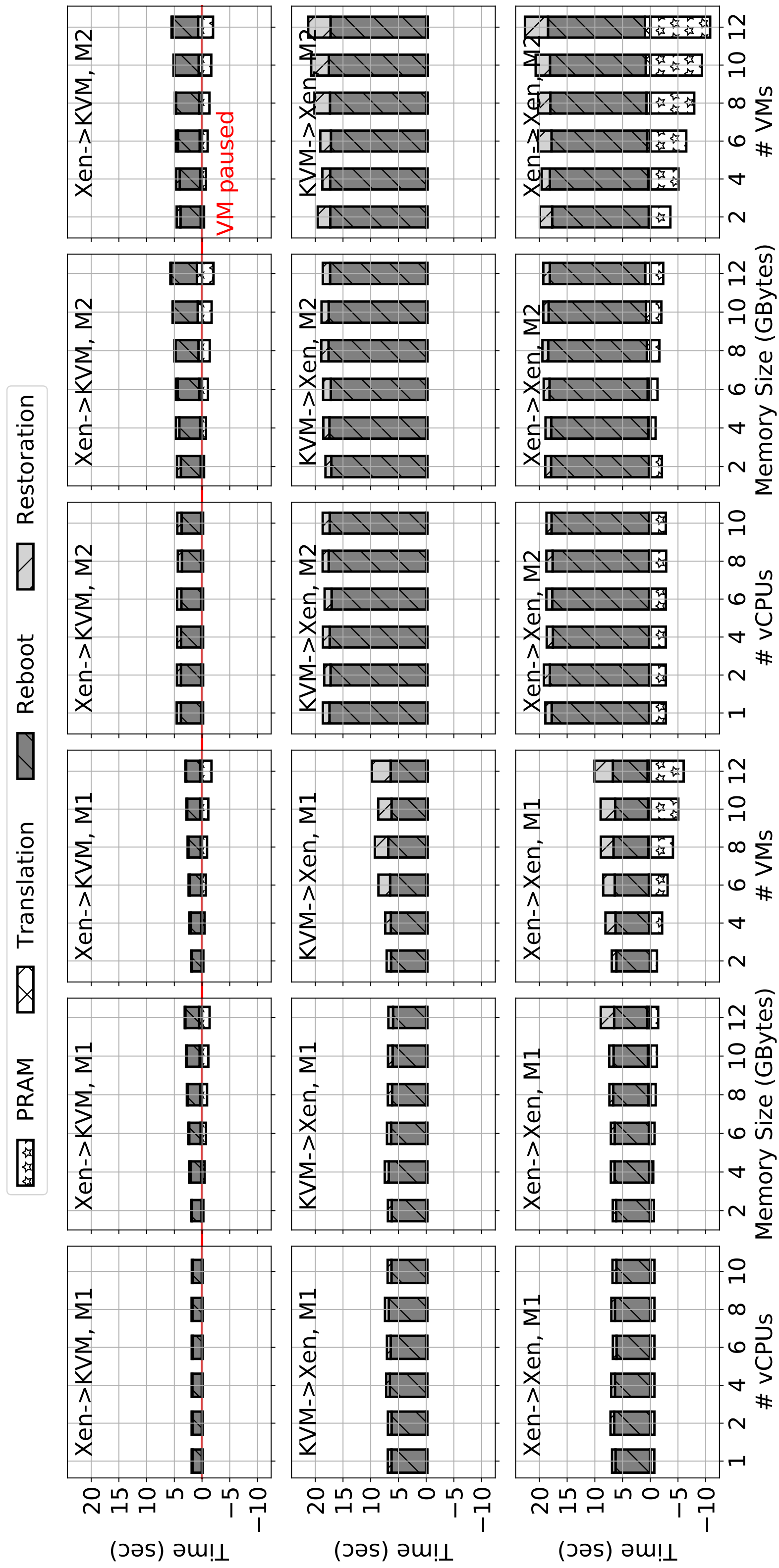

Figure 3.5: Scalability breakdown of InPlaceTP with regard to various transplantation directions and VM configurations.

migrating several VMs at the same time. This variation is explained by Xen's serialized migration process, which migrates multiple VMs in parallel on the sending side, but not on the receiving side. In particular, the first migrated VM's downtime will be lower than that of the second's, and so on. In comparison, MigrationTP offers a constant downtime on each VM by allowing multiple VMs to be migrated at the same time.

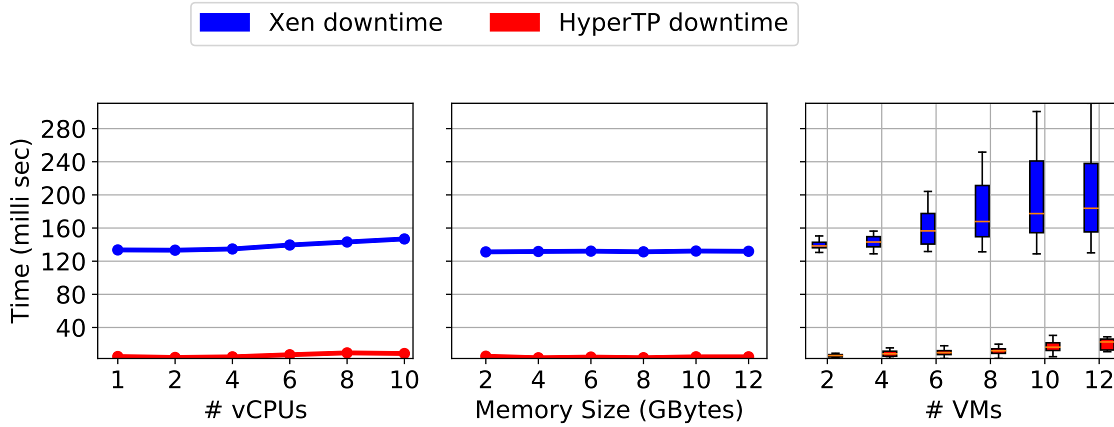

Figure 3.6: MigrationTP Xen→KVM downtime compared to Xen migration.

Figure 3.7 presents the total migration time of each solution. MigrationTP and Xen have almost the same results when migrating a single VM while varying its memory size (see the first two subfigures). Namely, while the number of vCPUs has no impact on the migration time, migration time scales almost linearly to VM memory size due to the need for transferring VM memory over the network. When varying the number of VMs, we observe that while MigrationTP has a higher median VM migration time, the variance in migration time is far less than that of Xen→Xen migration. This is again caused by Xen's serialized migration which blocks multiple VMs from being migrated at the same time.

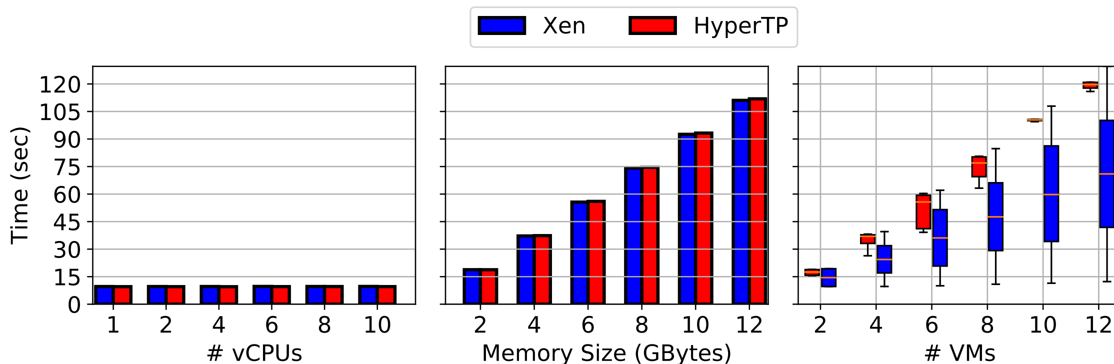

Figure 3.7: MigrationTP Xen→KVM migration time compared to Xen→Xen.

3.4.3 Impact on applications

We evaluated HyperTP using macro-benchmarks with common workloads in the following fashion: each benchmark is launched inside a Xen VM with 2 vCPUs and 8

GB of RAM; we then trigger the transplantation operation during each benchmarks' execution. We compare the results to that of a machine running purely on Xen, thus observing the impact of InPlaceTP on application performance.

Redis. We used the *redis-benchmark* tool to stress a Redis server running on a VM. The underlying host is then upgraded with HyperTP during the benchmark run.

Figure 3.9 presents the results for InPlaceTP on M1 and M2. The downtime of Redis is 8 seconds for Xen→KVM, 12s for KVM→Xen, and finally 13s for Xen→Xen on M1. On M2, the results are 11s for Xen→KVM, 22s for KVM→Xen and 23s for Xen→Xen. Note that this downtime includes the time needed to reestablish the physical network link on the host, which is done in parallel with other phases of InPlaceTP. While Redis continues to perform well after transplantation, we also observe a performance difference of approximately 16% between Xen and KVM for this particular workload.

Figure 3.8 (right) shows the Redis performance under MigrationTP, which like Xen→Xen migration, shows a "classical" live migration performance pattern with a performance drop during the memory copy phase (from 50s to 124s, or 78s in total), followed by a negligible downtime when the VM is paused, and finally a return to normal performance.

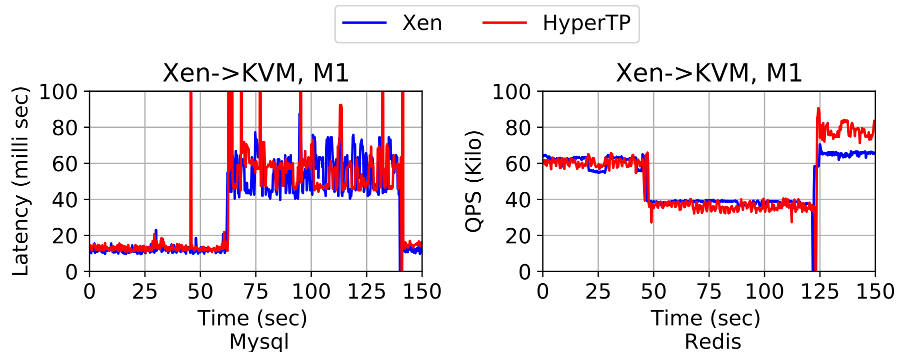

Figure 3.8: Impact of MigrationTP on MySQL (left) and Redis (right).

MySQL. We used Sysbench to generate load on a MySQL VM while applying HyperTP. With InPlaceTP, we observe a similar behavior as with Redis, where it causes a downtime of approximately 10 seconds on M1 and 21 seconds on M2 (see Figure 3.10). Both MigrationTP (Figure 3.8 left) and Xen→Xen migration caused a period of 252% increase in latency lasting 76 seconds during the migration process.

SPEC CPU2017. We ran all 23 SPECrate workloads included in the SPEC CPU2017 benchmark suite. We estimated the performance degradation caused by HyperTP as the maximum of the degradation w.r.t. Xen and KVM, i.e.

$$Deg = \max\left(\frac{t_{\text{HyperTP}} - t_{\text{Xen}}}{t_{\text{Xen}}}, \frac{t_{\text{HyperTP}} - t_{\text{KVM}}}{t_{\text{KVM}}}\right)$$

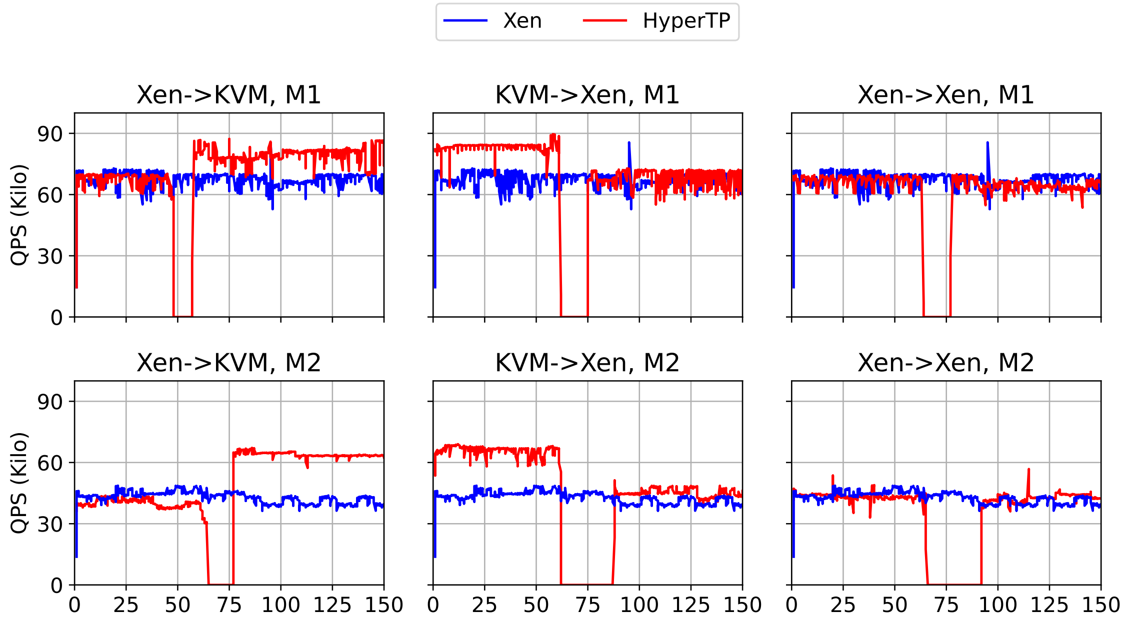

Figure 3.9: Impact of InPlaceTP on Redis throughput on M1 and M2.

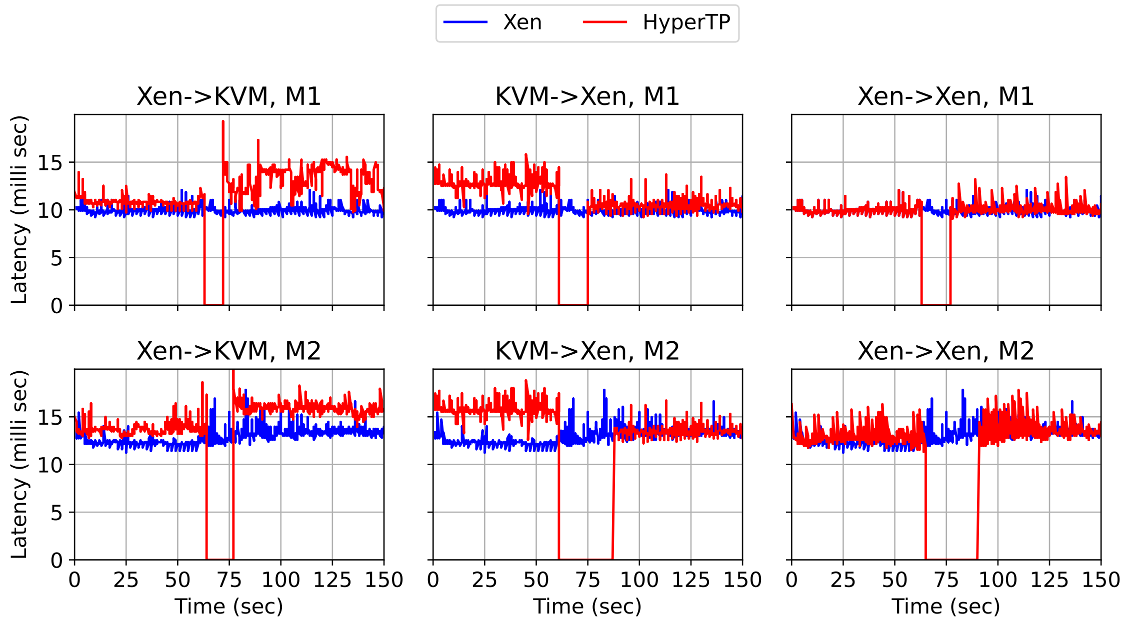

Figure 3.10: Impact of InPlaceTP on MySQL latency on M1 and M2.

Table 3.6: Impact of InPlaceTP and MigrationTP on SPECrate 2017 benchmarks.

					InPlaceTP						MigrationTP
	Xen		**KVM**		**KVM to Xen**		**Xen to Xen**		**Xen to KVM**		**Xen to KVM**
Benchmarks	Time M1	Time M2	Time M1	Time M2	Deg M1	Deg M2	Deg M1	Deg M2	Deg M1	Deg M2	Deg M1
perlbench	449.23	641.84	448.33	643.42	2.77	4.32	2.13	5.20	3.30	2.68	2.63
gcc	311.77	467.48	310.48	448.99	5.00	5.68	3.05	5.29	5.18	5.89	4.12
bwaves	939.70	1447.81	924.90	1420.64	2.92	3.74	2.33	3.70	1.65	1.00	1.83
mcf	450.80	712.02	449.33	664.51	4.04	3.88	2.82	4.01	3.45	1.49	3.12
cactuBSSN	312.87	524.55	310.01	507.50	5.12	4.14	4.45	4.03	3.58	2.39	3.82
namd	298.13	472.29	298.06	473.15	4.58	5.47	2.96	4.43	3.10	3.05	1.62
parest	636.41	1090.42	638.20	1088.68	2.28	2.29	1.47	2.46	2.24	1.39	1.15
povray	536.13	827.42	538.70	823.01	3.09	3.43	1.75	3.16	2.07	2.36	1.54
lbm	296.49	650.88	296.08	608.98	4.98	5.08	2.86	1.71	3.98	6.61	6.27
omnetpp	543.35	612.14	539.03	599.10	3.27	6.24	2.29	3.48	4.21	6.86	3.14
wrf	635.07	985.51	626.98	973.32	2.93	3.91	1.97	3.38	2.03	0.04	1.24
xalancbmk	473.08	625.59	472.07	617.82	2.40	5.28	1.47	5.00	6.68	5.69	0.79
x264	549.48	813.34	548.73	809.34	2.67	3.11	2.18	3.24	3.98	1.67	0.59
blender	423.02	582.61	422.51	580.16	3.41	4.59	2.16	4.14	3.05	2.64	1.68
cam4	524.44	811.05	522.12	807.74	3.35	3.48	2.14	2.33	2.81	2.38	2.24
deepsjeng	442.18	650.24	442.82	644.92	3.36	2.97	2.23	3.48	4.34	1.99	4.33
imagick	693.98	1008.05	693.31	1007.75	2.16	2.59	1.50	2.10	1.58	1.33	0.70
leela	725.80	1022.77	726.39	1018.50	2.10	2.82	1.26	2.56	1.55	1.73	0.65
nab	544.06	767.31	543.11	764.83	2.61	3.45	1.79	3.19	1.76	1.81	0.48
exchange2	566.03	856.35	565.98	857.11	2.44	2.99	1.63	2.70	1.73	1.67	0.96
fotonik3d	390.19	763.08	386.25	679.30	4.14	6.94	2.50	2.03	4.65	0.13	3.86
roms	422.13	662.71	417.98	656.53	4.41	1.67	3.78	4.16	4.89	0.31	5.76
xz	522.24	591.80	515.12	675.08	3.04	2.70	2.78	3.59	5.00	3.63	1.89

Table 3.6 presents each benchmark's execution time in seconds for Xen and KVM, as well as the performance degradation in percentage for each transplantation direction on M1 and M2. The maximum degradations for InPlaceTP are 5.12% on M1 and 5.00% on M2 for KVM→Xen, 4.45% on M1 and 5.29% on M2 for Xen→KVM, and 5.18% on M1 and 6.86% on M2 for Xen→Xen. MigrationTP's maximum degradation on M1 is 6.27%. Note that these differences not only come from the transplantation process itself, but also from the native performance difference between Xen and KVM. Indeed, we can see that these benchmark applications do not have the same performance in both hypervisors (see the Xen and KVM columns of Table 3.6). Moreover, since HyperTP's duration of performance degradation is quite constant, its impact on applications with longer execution times (e.g. scientific simulations) will be negligible.

3.4.4 Hypervisor update

In this section, we evaluated the time taken to upgrade a cluster using MigrationTP. We used the *BtrPlace* VM scheduler framework [47] to define the structure of a simple server cluster including 10 physical hosts. On each hypervisor host, we defined 10 VMs each equipped with 1 vCPU and 4 GB of RAM, for a grand total of 100 VMs across all hosts. In this group of VMs, we configured 30% to run a video streaming server (each with a matching client running outside of the cluster); 30% running a CPU- and memory-intensive benchmark; and the remaining 40% being idle. We simulated an upgrade event by dividing the cluster into smaller groups, sequentially putting each group offline using *BtrPlace*'s constraints (placing VMs from the offline group into other groups), followed by recording the resulting migration plans. *BtrPlace* generated a migration plan with a total of 154 VM migration operations. We then prepared a real software cluster running the above-defined VMs, executed the migration plans proposed by *BtrPlace*, and recorded the total migration times of the cluster.

We repeated the same experiments while varying the percentage of VMs that are InPlaceTP compatible (i.e. VMs on which the characteristics of InPlaceTP are acceptable, and therefore do not need to be migrated). Figure 3.11 shows the number of migrations and reduction in total migration times (compared to normal migration-based upgrades) with varying proportions of InPlaceTP-compatible VMs. We observe that increasing the proportion of InPlaceTP-compatible VMs reduces the number of migrations necessary to upgrade the cluster, as well as the total migration times. For example, with 20% InPlaceTP-compatible VMs, our migration plan required 109 migrations, corresponding to a 17% shorter migration duration. With 60% compatible VMs, the cluster needed 73% fewer migrations and 68% less migration time, and with 80% compatible VMs, the cluster required only 25 migrations, or reducing total migration time by almost 80%. Coupled with the fact that InPlaceTP takes only seconds to complete, these results show how HyperTP can substantially speed up the upgrading of a hypervisor cluster.

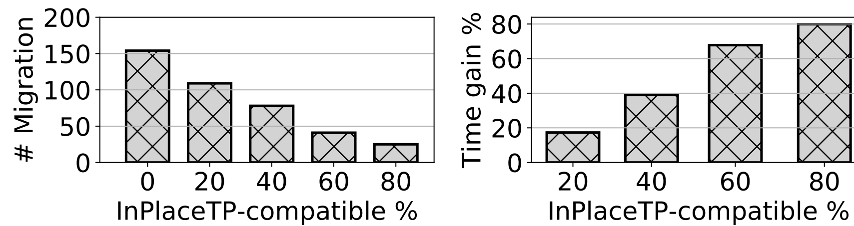

Figure 3.11: Impact of InPlaceTP on cluster updating: a) w.r.t. the number of migrations; b) w.r.t. total update time.

3.4.5 Hypervisor security

As we claimed in Section 3.2.4, HyperTP can be used to reduce the vulnerability window of a hypervisor. To qualify our claim, we study the case of a vulnerability on Xen. For example, consider CVE-2018-18883 [48], a critical denial-of-service vulnerability with a CVSS v2 score of 7.2 affecting Xen 4.9.x-4.11.x on Intel x86 platforms. In short, the vulnerability allows Xen HVM guests to configure CPU virtualization features even if they are disabled in its configuration, leading the host Xen to access uninitialized memory and causing Xen to crash. The root cause of CVE-2018-18883 is mismanagement of Intel's virtualization features; hosts running KVM are not vulnerable to the same issue. In this case, HyperTP can help quickly handle such a vulnerability by switching from Xen to KVM.

Another use case of HyperTP is to apply certain *software diversity*-based security measures on-the-fly during operation of an hypervisor. For example, certain operating systems support link-time randomized binary layouts [49, 50]. However, these defensive measures can only be applied once at boot-time, meaning they become less effective over time in long-running systems. HyperTP can be used to periodically reapply these measures by switching from H_{current} to a H_{target} with a different random binary layout, making attacks requiring running the same binary for a long time (e.g. those that need to probe the address space) more difficult to execute.

3.4.6 Memory overhead

The memory overhead of HyperTP includes the extra memory required for storing PRAM structures and UISR states. Figure 3.12 presents our overhead measurements for various transplantation scenarios as explored in Section 3.4.2. We can see that the memory footprint of PRAM structures increases with the VM memory size, from 16 KB (for a single 1 GB VM) up to 60 KB (for a 12 GB VM). In the case of multiple simultaneously-running VMs, the overhead increases slightly due to additional file info and metadata pages needed for each VM (see Figure 3.3); however, these overheads remain minimal at only 148 KB for 12 VMs with 1 GB of RAM each. More generally, PRAM structures consist of 8-byte records for every VM's memory page (which can be 4K or 2M in size) leading to a worst-case overhead of 2 megabytes of metadata per GB of guest memory (in the case of all-4K guest pages), or 4 KB per GB of guest memory

(in the case of all-2M guest pages).

The memory footprint of UISR states increases with the total number of vCPUs, from 5 KB with 1 vCPU up to 38 KB with 10 vCPUs. In summary, the total memory overhead of HyperTP varies from 21 KB up to 98 KB per VM, which is negligible. Note that this extra memory is immediately given back to the hypervisor as soon as the transplantation process finishes.

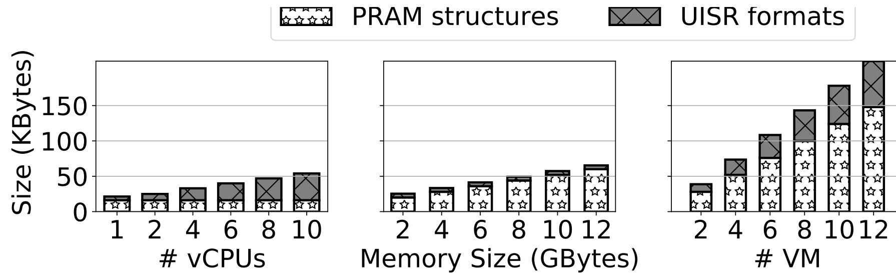

Figure 3.12: Memory overheads of InPlaceTP and MigrationTP.

3.5 Discussion

UISR and VM compatibility. Section 3.2.1 specified that UISR serves as a representation of VM state that is sufficient for its reconstruction. However, restoration of a VM from a given UISR instance is affected by several factors: (1) hardware compatibility between the source and target hypervisor platforms, e.g. matching processor features; (2) each hypervisor has numerous different implementations of devices and resources, some of which cannot be easily reconstructed (e.g. passthrough devices); and (3) breaking changes to the hypervisor's paravirtualization API contract. As a result, the set of features made available to the VM must be chosen such that either (1) the feature could be reproduced in our UISR and in the target hypervisor; or (2) the target VM is tolerant to loss of said feature's states. In our implementation, devices that fall into the second category (e.g. PCIe network devices) are implemented using the stop-reconnect technique, as the guest Linux operating system in combination with our kernel driver allows removing and reinstalling these devices without affecting network connections.

Downtime-resource tradeoff. As stated above, datacenter operators must choose whether InPlaceTP or MigrationTP is more appropriate for the maintenance of their virtualization platform. While our evaluations show that InPlaceTP shortens the maintenance duration, it also comes with an intrinsically longer downtime (dominated by the booting of the new hypervisor, as seen in Section 3.4.2). The individual downtime of each VM (several seconds) might not be acceptable compared to the milliseconds of downtime offered by migration-based techniques. Moreover, this downtime tends to slightly increase along with the number of VMs; this increase, while predictable, must be taken into account during the maintenance planning. One way to perform this is by mixing and matching the two techniques: for instance, VMs can be separated into

service tiers; as an example, one can define two tiers denoted “default” and “mission-critical”. Most VMs that can tolerate a short downtime can be placed in the “default” tier on machines serviced using InPlaceTP; VMs denoted “mission-critical” can instead be hosted on machines using MigrationTP, where upgrading will not cause a significant disruption.

Hypervisor security. It is worth noting that in the context of hypervisor security, HyperTP first of all serves as a *mitigation*, meaning it is applied *after* a vulnerability becomes known, as illustrated in Figure 3.2. Moreover, before the mitigation could be applied, the vulnerability must be analyzed to determine which HyperTP-capable hypervisor is an appropriate transplant target (e.g. not vulnerable to the vulnerability in question). Secondly, HyperTP cannot currently serve as a replacement for remediation of a compromised system. Note that techniques exist to bring a running system to a known, verifiable state (e.g. Dynamic Root for Trusted Measurement (DRTM) [51, 52]) and can potentially be used to reinforce such a remediation. To elaborate, a DRTM launch event can be used to implement micro-reboot by taking over a running hypervisor, putting the system in a known trusted state, then booting the target hypervisor. As the target hypervisor started from a trusted state, it is considered “fresh” and free from any leftover compromised state. It can then pick up the UISR left by the running hypervisor and resume any running VMs. However, a limitation of InPlaceTP is that it currently does not implement these techniques. In comparison, MigrationTP and other live migration-based techniques [9, 53] can be used to transfer VMs from a potentially-compromised host to another hypervisor. While none of these techniques (whether micro-reboot-based or migration-based) can guarantee the integrity of the aforementioned VMs post-compromise, this limitation can again be mitigated by the use of encrypted virtual machines (e.g. AMD SEV [54]).

3.6 Related works

Our investigation of the state of the art will focus on the applications of HyperTP on hypervisor update and security.

3.6.1 Hypervisor update

Live patching. The least disruptive method for updating the hypervisor is kernel live patching [55, 56]. Live patching is a lightweight solution for applying simple temporary patches to a running kernel. Unfortunately, it does not support patches that may change persistent data structures (i.e. data structures which have allocated instances in the kernel heap or stacks). When such patches are not sufficient, VM live migration or in-place hypervisor update with server reboot should be used instead.

Live migration. VM live migration allows the cloud provider to upgrade almost everything on the origin server, from hardware devices to the hypervisor, once it no

longer hosts any running VMs. Several works [57, 58] have investigated downtime reduction during live migration. Tsakalozos et al. [59] proposes the use of a special-purpose *MigrateFS* file system and a network of brokers for synchronizing virtual disk states to ease the migration of VMs without making use of shared remote storage. These approaches can also be combined with MigrationTP to further improve the performance of migrating VMs that are not compatible with InPlaceTP.

To our knowledge, Liu et al. [60] is the only work which studied VM migrations between heterogeneous hypervisors as HyperTP. It was not possible for us to quantitatively compare our MigrationTP solution with Liu et al. [60] because no public prototype exists. From the design perspective, our UISR principle facilitates the integration of new hypervisors, making HyperTP generic. Finally, HyperTP combines live migration with in-place hypervisor transplantation to address the scalability limitation of the former.

In-place hypervisor update. Zhang et al. [9] introduces *Orthus*, which targets the upgrading of both the user-space emulator software (QEMU) and the KVM kernel module with minimal downtime. Orthus modifies the KVM module to incorporate state-transition capabilities between two consecutive versions, coupled with a lightweight mechanism to checkpoint/restore VMs. However, *Orthus* is specific to KVM, and does not target heterogeneous hypervisors like HyperTP. Secondly, *Orthus* does not target the update of the entire kernel, which explained their very low downtime (0.48-9 seconds). Other research works such as [40, 53, 61] uses nested virtualization to enable quick and transparent in-place updates. LivCloud [62] solves the related problem of migration compatibility between different cloud providers by making use of a common L1 hypervisor. While these works are comparable with MigrationTP in that they also propose a low-downtime solution for updating the L1 (i.e. “inside”) hypervisor, they did not propose a mechanism for updating the L0 (i.e. “outside”) hypervisor. HyperTP, in comparison, does not include this limitation as the hypervisor kernel is entirely restarted. Nested virtualization in this fashion also incurs an additional overhead, especially on the commonly-used x86 architecture where virtualization instructions executing inside the L1 hypervisor must be trapped and emulated by the L0 hypervisor.

3.6.2 Hypervisor security

We classify hypervisor protection strategies in four categories: preventive (stopping attacks by design), corrective (applying updates), reparative (restoring consistency), and defensive (protection during the vulnerability window).

Preventive approaches, e.g. hardening the hypervisor: Many research works advocate a micro-kernel architecture for the hypervisor in order to (1) reduce the trusted computing base (TCB), thus reducing the attack surface [63, 64]; (2) formally verify this TCB to prove the absence of known classes of vulnerabilities [65]; and (3) isolate buggy or untrusted device drivers of the hypervisor [63, 66, 67]. This approach often imposes a strict implementation of a micro-kernel architecture, which

requires considerable efforts in the hypervisor's design and implementation. In addition, most of the contributions in this approach require hardware changes that are not yet available [68]. Moreover, no implementation is 100% sure; such an approach has to be combined with regular security updates as studied in the next section.

Preventive approaches, e.g. software diversity: The concept of software diversity involves using multiple software versions to mitigate vulnerabilities. Schaefer et al. [69] describe general approaches for utilizing and managing diverse software systems; In the context of virtualization, Winarno et al. [70] studies the use of multiple hypervisors to ensure system resilience. Tan et al. [71] proposes a similar scheme based on one hypervisor running on multiple cloud platforms. However, to our best knowledge, our work is the first that allows a VM to be transplanted between multiple hypervisors on the same machine with minimal disruption.

Reparative approaches, e.g. consistent state restoration: These mainly rely on fast reboot and restoration, and can be implemented at the OS or hypervisor level [72, 73, 74, 75, 76, 77, 78]. Notably, Otherworld [73] restores applications running on a kernel in the event of a crash by booting a previously-loaded second kernel image, and restoring the application from main memory. The authors of [72, 74] went in the same direction with hypervisors by saving the states of VMs in memory and restoring them to a new loaded hypervisor on the same server. Cerveira et al. [78] proposed to respond to hypervisor corruption by migrating VMs over the same physical host instantly and with no overhead, by avoiding memory copy and taking advantage of Intel EPT's inner workings.

Defensive approaches, e.g. mitigation during vulnerability windows: The aforementioned approaches have a limitation: that **they cannot protect against a vulnerability if the corresponding security patch is not yet available**. As we highlighted in Section 3.2.4, the creation of such a security patch can take anywhere from several days to multiple months. In contrast, HyperTP provides an unique "escape hatch" that protects virtualization infrastructure during a vulnerability window with little downtime (as long as the vulnerability does not impact the target hypervisor). The combined approach of HyperTP also gives operators a flexible tradeoff between downtime and resource usage.

3.7 Summary

We introduced HyperTP, a platform for replacing a running hypervisor with a different hypervisor while incurring minimal downtime in a process called *hypervisor transplant*. We discussed our ideas of *VM state hierarchy* and *Unified Intermediate State Representation*, and presented details of the two approaches that make up HyperTP, in-place hypervisor transplant (InPlaceTP) and migration-based transplant (MigrationTP). We

evaluated our prototype of HyperTP with well-known benchmarks, and showed that HyperTP causes minimal interference to running workloads. Namely, InPlaceTP needs less than 2 seconds to transplant a VM running on Xen to KVM, while requiring negligible memory and I/O overhead. MigrationTP transplants a VM to another hypervisor with essentially the same cost as normal live migration. We showed that a hypervisor cluster with 80% of VMs supporting InPlaceTP can reduce its upgrade time by a proportional 80%, and demonstrated how HyperTP can simplify the process of securing hypervisor infrastructure.

This chapter was originally published as: Tu Dinh Ngoc, Boris Teabe, Alain Tchana, Gilles Muller, and Daniel Hagimont. HyperTP: A unified approach for live hypervisor replacement in datacenters. *Journal of Parallel and Distributed Computing*, page 104733, 2023. doi:10.1016/j.jpdc.2023.104733.
© 2023 Elsevier Inc. Reprinted under permission.

Chapter 4

Optimized resource allocation on virtualized non-uniform I/O architectures

Contents

4.1 Overview 46

4.2 Impact of NUIOA on VM performance 47

4.2.1 Methodology 47

4.2.2 Experimental results 49

4.2.3 Lessons learned 52

4.3 NUIOA-aware VM resource allocation strategy 53

4.3.1 Implementation 54

4.3.2 Discussion 55

4.4 Evaluation 55

4.4.1 Result analysis 56

4.4.2 Scalability evaluation 59

4.4.3 Overhead of NUIOA-aware allocation 60

4.5 Related works 61

4.5.1 Software solutions 61

4.5.2 Hardware solutions 62

4.5.3 Positioning of our work 62

4.6 Summary 62

4.1 Overview

To recall Chapter 2, hardware features such as the *I/O memory management unit* (IOMMU) grant VMs direct access to a physical device, in a process called *device passthrough*. When supported, device passthrough provides VMs with the lowest virtualization overhead and the best possible performance, and therefore is the focus of our current work. On current server platforms, hardware components are often connected to each other via a PCIe bus. Notable components connected using the PCIe bus include network adapters, storage cards/adapters, GPUs and so on; PCIe has even been adapted as a cluster interconnect bus, as specified by the *Compute Express Link* (CXL) standard [79].

Current OSes and hypervisors already support NUMA-aware scheduling [80, 81], i.e. they take into account NUMA distance costs (between memory and CPU) when scheduling multiple threads. However, NUIOA scheduling is far less often explored; while hypervisors are capable of acquiring home node information for each device (e.g. by using the proximity domain information exposed by ACPI), they do not use this information by default when scheduling VMs. As a consequence, a VM configured with device passthrough might be scheduled on a remote node, leading to lower performance than one might expect. Thus, two VMs with the same characteristics may have different I/O performance simply because one VM has direct access to the device's home node and the other not, leading to a *performance unpredictability*. This performance unpredictability is undesirable both in terms of impacting the SLA provided to consumers, whether internal (for private clouds) or external (for public clouds), and in terms of resource utilization and VM density. Certain solutions like OpenStack [82] make efforts to optimize their VMs for NUIOA; however, they are limited to scheduling whole guests on the corresponding home node, without targeting the applications that live inside these VMs [83]. It is therefore desirable to have an allocation strategy that takes into account NUIOA effects and remedies the issues coming from remote I/O operations.

Our contribution in this chapter is twofold: (1) we carry out an exhaustive study of NUIOA impact on VM I/O performance; and (2) based on our findings, we propose a novel VM resource allocation strategy on NUIOA systems.

Study of NUIOA impact on performance. We first carried out I/O performance evaluations of VMs hosted by the Xen hypervisor. We experimented with various I/O workloads with different resource allocation configurations under two types of connections, namely Ethernet and InfiniBand. We focused on three different aspects of performance impacts caused by NUIOA architectures: (1) latency impacts caused by remote NUIOA accesses; (2) interconnect bandwidth limitations caused by neighboring workloads; and (3) the overhead caused by virtual NUMA configurations on VM performance. We found that as expected, I/O performance on VMs is optimal when they are located on the corresponding device's home node; moreover, the impact of NUIOA is especially relevant concerning high-speed networking, especially when an efficient I/O stack is used.

NUIOA-aware VM allocation scheme. Our second contribution is a NUIOA-aware

VM allocation scheme (*NUIOA allocator* for short) for combating NUIOA effects. The basic idea behind our NUIOA-aware allocator is threefold. Firstly, we ensure that each VM assigned with a device is also provided with one part of the home node's CPU and memory resources. Secondly, we inform each VM of this association between device and resources by exposing a virtual NUMA (vNUMA) topology. Finally, we optimize the scheduling of I/O applications in VMs by locating them on home nodes to avoid any NUIOA effects while avoiding oversubscribing or wasting resources. We experimented with our NUIOA-aware allocation strategy on Xen; our results show that the NUIOA allocator improves application performance by up to 20% compared to resource allocation strategies in Xen.

The remainder of this chapter is organized as follows. Section 4.2 presents an evaluation of an I/O heavy application with multiple different NUIOA setups; from our evaluation, we observe which factors influence NUIOA effects, and therefore I/O virtualization performance. Section 4.3 builds on our aforementioned observations to establish our contribution, a NUIOA-aware VM resource allocation strategy that involves a hypervisor-layer NUIOA allocator and a workload scheduling methodology. We also present the implementation of our resource allocation strategy in detail, and provide some discussions on potential venues for improvement. In Section 4.4, we show how our NUIOA-aware VM resource allocation strategy minimizes the NUIOA penalty of virtualized I/O in comparison to various other setups, and discuss the potential overhead of our solution. In Section 4.5, we study existing software and hardware solutions that tackle the NUIOA problem, and discuss the various tradeoffs relevant to each solution. Finally, Section 4.6 concludes the chapter.

4.2 Impact of NUIOA on VM performance

4.2.1 Methodology

In this section, we first evaluate various I/O workloads to see how NUIOA impacts application performance.

Hardware setup. Experiments run on a cluster of Dell PowerEdge R630 servers, the configurations of which are detailed in Table 4.1. Each server is equipped with two processors, and thus divided into two NUMA nodes numbered 0 and 1; both the Ethernet and InfiniBand devices are connected to node 0 of each server, or the **home node**.

Table 4.1: Hardware configurations used for NUIOA evaluations.

Component	Characteristics
CPU	2x Intel Xeon E5-2630 v3 (8 cores per node)
Memory	128 GB (64 GB per node)
Ethernet adapter	Intel 82599ES 10 Gbps (node 0)
InfiniBand adapter	Mellanox MT27500 56 Gbps (node 0)
Storage	600 GB HDD

Software setup. To measure I/O performance, we use the Sockperf benchmark tool [84]. We run Sockperf in one of two modes depending on what we wanted to examine: "latency under load" mode measures packet latency under a certain network load level, and therefore is affected by NUIOA latency; and "throughput" mode measures the maximum throughput delivered by the network card and is therefore affected by NUIOA bandwidth effects. For the purposes of virtual NUIOA evaluations, we deploy Sockperf on a VM running Ubuntu 20.04 on top of Xen [85] version 4.11. Each VM is equipped with 6 vCPUs, 8 GB of memory, and two network interfaces: one Ethernet and one Infiniband, both working in device passthrough mode. For each benchmark, the VM connects to a secondary machine with the same hardware configuration running bare-metal Linux.

Experiment details. As explained in Section 4.1, NUIOA effects originate from the need for I/O operations to cross the NUMA interconnect. To see how these effects influence I/O-heavy applications, we set up the following experiments:

1. **NUIOA latency effects.** In this experiment, we run Sockperf in latency mode while varying the locality of the VM with regards to the network device's home node in order to investigate how NUIOA affects I/O latencies.

2. **NUIOA with bandwidth contention.** In this experiment, we discover how competing NUMA interconnect traffic influences NUIOA effects on application performance. We use a competing *neighbor* application to add load to the interconnect.

3. **vNUMA configurations.** In this experiment, we study the impact of NUMA configuration for the VM on I/O performance with NUIOA.

We define the configurations we used for the aforementioned experiments in Table 4.2. Each row of the table corresponds to a distinct VM resource configuration. For example, in the first listed configuration named A^0_{Eth}, we provision a VM with its resources allocated on physical node 0 and a passthrough Ethernet adapter, with no interfering neighbor; moreover, we allocate the privileged domain (Xen Dom0) entirely on node 0. Note that since our VMs use the devices in passthrough mode, the privileged domain does not interfere in the communication path with the devices. Thus, all the results obtained would have been similar if the privileged domain resources were allocated on node 1.

For each configuration, we measure the median network latency with Sockperf in both directions: reception (Rx) and transmission (Tx) while varying the network load (message size and rate). For the evaluation on Ethernet, we use the default Linux TCP stack as it is the case with most applications. With InfiniBand configurations, we set up Sockperf with libvma 9.0.2 [86] and MLNX_OFED 4.9 drivers to take advantage of the provided kernel-bypass features as is often done for HPC workloads. We repeat each experiment five times; as we observe low standard deviation among these runs, we report the average results.

Table 4.2: Various configurations and associated acronyms (see Section 4.2.2)

Name	Node	Device	Dom0	Neighbor
A^0_{Eth}	Node 0	Ethernet	Node 0	No
A^0_{Ib}	Node 0	Infiniband	Node 0	No
A^1_{Eth}	Node 1	Ethernet	Node 0	No
A^1_{Ib}	Node 1	Infiniband	Node 0	No
N^0_{Ib}	Node 0	Infiniband	Node 0	Yes
$\upsilon N^{0,1}_{\text{Ib}}$	Nodes 0 & 1	Infiniband	Node 0	No
$U^{0,1}_{\text{Ib}}$	Nodes 0 & 1	Infiniband	Node 0	No

4.2.2 Experimental results

We consider one ideal configuration (A^0_{Eth} on Ethernet and A^0_{Ib} on Infiniband) as the reference configuration *ref*; for each other configuration X, we calculate the performance impact *deg* using the following formula for latency (*lat*) and bandwidth (*bw*):

$$deg_{\text{lat}} = \frac{lat_X - lat_{\text{ref}}}{lat_{\text{ref}}} \tag{4.1}$$

$$deg_{\text{bw}} = \frac{bw_{\text{ref}} - bw_X}{bw_{\text{ref}}} \tag{4.2}$$

with lat_X and bw_X being the latency and bandwidth results of configuration X, respectively.

NUIOA latency effects. To assess the impact of the NUMA interconnect distance on I/O performance, we compare the network latency and bandwidth figures obtained when a single VM is allocated on the home node versus when it is allocated on the remote node. This corresponds to comparing the performance of A^0_{Eth} versus A^1_{Eth} for Ethernet devices, and A^0_{Ib} versus A^1_{Ib} for the InfiniBand device respectively. Figure 4.1 presents NUIOA latency impacts on both Ethernet and InfiniBand devices; similarly, Figure 4.2 presents NUIOA bandwidth impacts.

For Ethernet devices (first row of Figure 4.1), we observe that these devices experience relatively consistent NUIOA performance degradation, at around 7% for both transmission and reception regardless of packet rate. This is explained by the longer I/O path of Sockperf with Ethernet devices, which involves the Linux kernel's network stack and therefore serves to mask NUIOA effects. In fact, without a kernel-bypass solution, Sockperf on Ethernet shows little to no bandwidth impact from NUIOA at any setting (see Figure 4.2a).

Concerning CPU usage with Ethernet (Figure 4.3), we observe a small (approximately 5%) but consistent difference in CPU consumption between the two configurations at packet sizes higher than 32 bytes on the receive path; this can be explained by the higher cost of cross-node memory copies needed for packet reception. Conversely, we observe fairly equal CPU consumption between the two configurations on the transmit path, with higher CPU usage while sending 32-byte packets.

In the case of InfiniBand networking, Sockperf communicates directly with the device through the use of libvma-based kernel-bypass; the optimized I/O path of

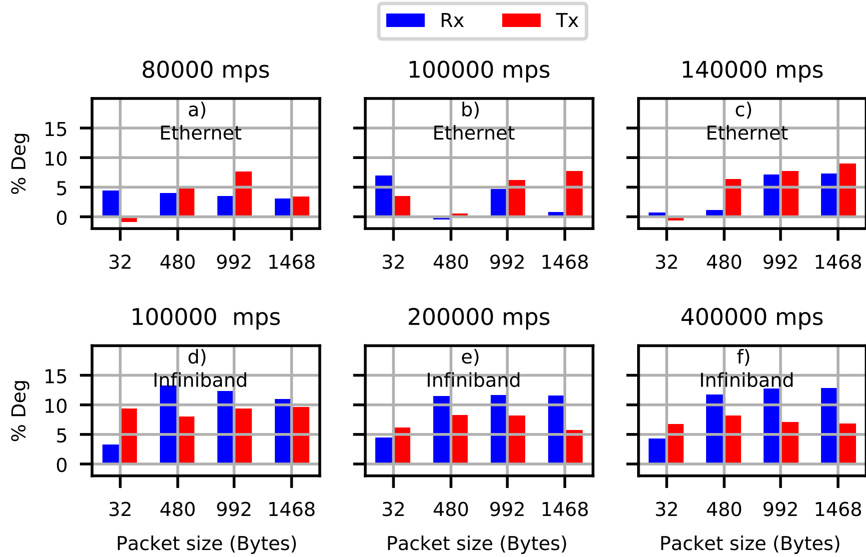

Figure 4.1: NUIOA latency degradation percentage (% Deg) on Sockperf performance for A^0_{Eth} versus A^1_{Eth} and A^0_{Ib} versus A^1_{Ib} in function of messages per second (mps) and packet size. Rx and Tx stand for reception and transmission respectively.

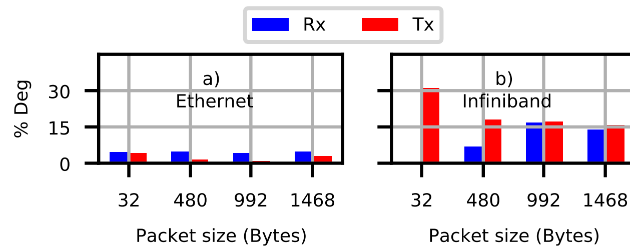

Figure 4.2: Socket-interconnect impact on Rx/Tx bandwidth.

libvma allows us to observe NUIOA effects in more detail. Firstly, we observe that Sockperf's CPU consumption stays the same under all configurations, since Sockperf on InfiniBand makes use of active polling to minimize latency. Note that high-speed Ethernet devices are still subject to the same NUIOA effects, especially with optimized I/O stacks like DPDK (Data Plane Development Kit) or libvma (Mellanox's Messaging Accelerator). In general, we observe an approximately 13% latency degradation on the receive path, compared to 6-10% on the transmission path. This is explained by an extraneous memory copy on the Rx path resulting from zero-copy receives being disabled in Sockperf. This justifies the lower Rx performance impact with 32-byte packets. Regardless, we still observe an effect on packet latency on the Tx path, which holds true regardless of message size and rate, suggesting that this impact is inherent to NUIOA and not dependent on memory bandwidth usage.

NUIOA with bandwidth contention. In this experiment, we aim to see whether neighbor VMs with heavy memory activity impacts the performance of I/O applications. We focus on evaluating the performance of InfiniBand devices to emphasize NUIOA effects as discovered in the previous experiment. We prepare a neighbor VM that

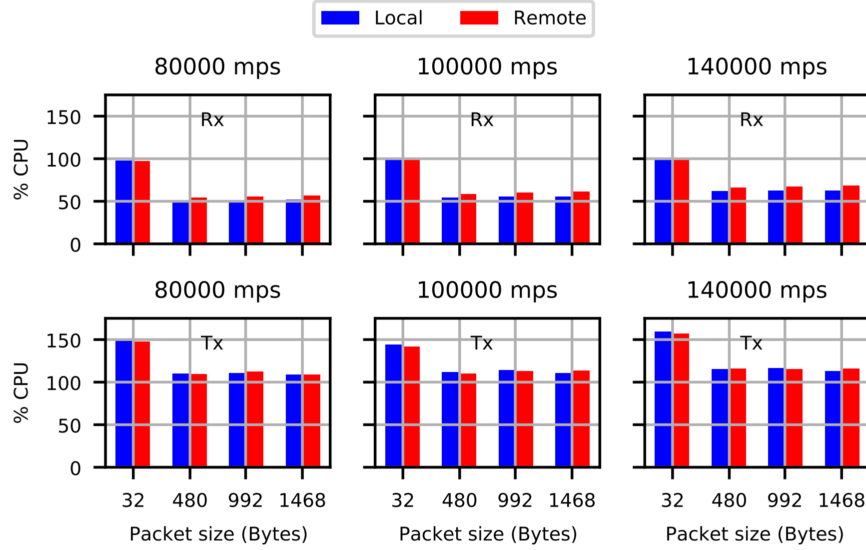

Figure 4.3: CPU consumption for A^0_{Eth} versus A^1_{Eth} on Ethernet.

executes Sysbench [87] in memory mode using 8 threads; we configure this neighbor VM to be located on the opposite node of the VM's node, so that it utilizes interconnect bandwidth without polluting the VM node's CPU caches or interfering with CPU scheduling. We compare this setup with a setup without a neighbor VM (N^1_{Ib} and A^1_{Ib} in Table 4.2). Figure 4.4 shows the resulting network performance. We observe only 1-2% performance variability for both the Rx and Tx datapath; this shows that the memory load levels generated by Sysbench does not have a significant effect on I/O applications.

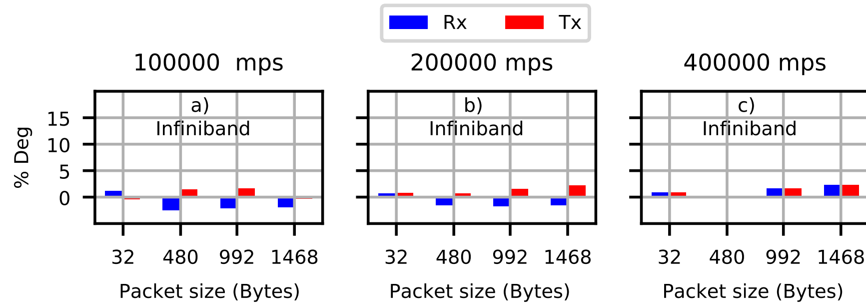

Figure 4.4: Impact of a neighbor VM on Sockperf performance.

vNUMA configurations. VMs that span multiple physical NUMA nodes are typically unaware of the underlying physical topology, and therefore are not able to make appropriate scheduling decisions. Hypervisors can remedy this issue by using one of two approaches: (1) interleaving the memory of the VM between the allocated nodes and presenting a UMA architecture to the VM (the default on Xen); or (2) exposing a virtual NUMA topology that mirrors the underlying topology used to allocate the VM's resources. In our experimental configurations, we allocate 50% of the VM's resources on each physical NUMA node.

We start by comparing the default interleaved configuration $U^{0,1}_{\text{Ib}}$ (i.e. without

vNUMA) to the reference configuration A^0_{Ib}. The first row of Figure 4.5 shows the recorded latency. We observe nearly the same or even worse performance compared to the previous experiment (13% for Rx and 16% for Tx) even though parts of the VM are located in the network device's home node. This is explained by the VM not having knowledge of NUMA topologies nor of device locality, resulting in I/O applications being scheduled on a remote node, combined with remote NUMA accesses caused by the lack of memory/CPU locality.

Next, we study Sockperf performance on VMs with vNUMA enabled (see configuration $vN^{0,1}_{\text{Ib}}$ of Table 4.2). We configure our VM with two virtual NUMA nodes, each equipped with 4 GB of RAM and 3 vCPUs. We present our results in the second row of Figure 4.5. We observe no performance degradation on the Rx path compared to the reference configuration, since the VM is capable of utilizing the vNUMA topology to avoid remote memory accesses during packet copies; however, the Tx path shows significant performance degradation of up to 20%, in the same fashion as seen in the UMA configurations presented above. We also observe that in this configuration, neither Xen nor the guest take into account NUIOA effects while scheduling tasks; as a result, the Tx thread of Sockperf is occasionally scheduled onto a remote node, causing performance degradation.

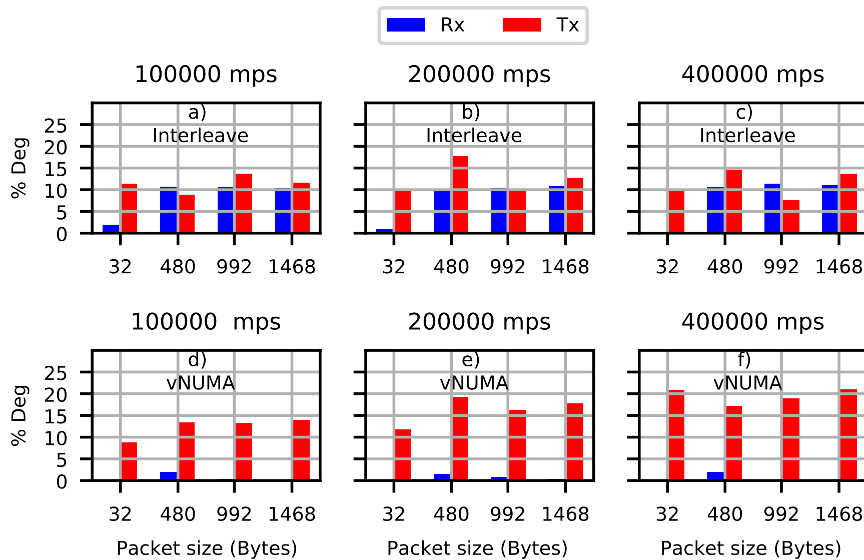

Figure 4.5: Impact of interleaved ($U^{0,1}_{\text{Ib}}$) and vNUMA ($vN^{0,1}_{\text{Ib}}$) configurations on Sockperf latency compared to the reference A^0_{Ib}.

4.2.3 Lessons learned

Following the aforementioned experiments, we list our observations of I/O application performance running on VMs under NUIOA conditions.

1. As expected, I/O applications reach maximum performance when they are located on the corresponding device's home node.

2. Performance impacts caused by NUIOA effects are most notable with high-speed I/O, such as InfiniBand devices with kernel-bypass libraries.

3. NUMA interconnect utilization by neighboring workloads has little impact on I/O performance.

4. While a UMA VM configuration can cause performance degradation of I/O applications, vNUMA can serve to improve performance under certain conditions (as seen in the vNUMA Rx benchmarks).

4.3 NUIOA-aware VM resource allocation strategy

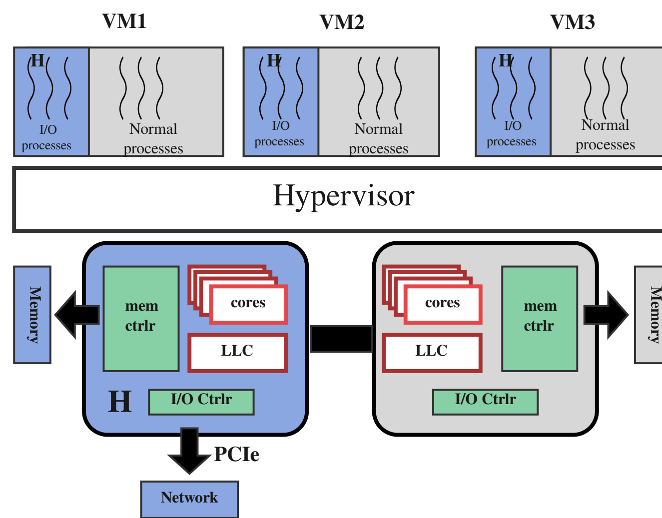

Figure 4.6: Overview of our NUIOA-aware allocation strategy. I/O processes of each VM are allocated on the home node H (colored in blue)

In this section, we present our resource allocation approach for mitigating the impacts of NUIOA effects on I/O workloads.

Overview. Our allocation strategy relies on the observation that ideally, the best resource allocation strategy for I/O applications is to place all of them on their corresponding home node. However, CPU and memory constraints (e.g. physical core counts and memory module sizes limits) mean that this strategy is not always applicable in practice. We work around this issue by ensuring that each aforementioned VM is allocated parts of the resources belonging to its device's home node. In other words, if a network device is connected to node H then any VM with I/O applications using that device would receive parts of the resources of node H, as illustrated in Figure 4.6. Additionally, we ensure that I/O workloads are scheduled on the correct node for its device to avoid any associated NUIOA costs. We describe below our allocation strategy in detail.

Hypervisor-layer resource allocation. Our first contribution is a hypervisor-side allocator that ensures I/O-heavy VMs are provided with resources backed by an appropriate NUMA node. We first gather information about the system's NUIOA topology,

including the relationship from each PCI device to the NUMA node it is attached to. The system administrator defines each VM as being I/O-heavy by configuring device passthrough for that VM. For each such VM, we configure its resources allocation using vNUMA to provide it with parts of the NUMA node that its passthrough device is attached to. If the VM is not declared as an I/O-heavy VM, we simply fall back to the default allocation behavior. Otherwise, given a VM with a memory size of m and n vCPUs, we calculate the memory and CPU resources m_H and n_H allocated to the VM on its home node H using the following formulas:

$$m_H = \left\lfloor \frac{M_H^f \times m}{M^f} \right\rfloor \tag{4.3}$$

$$n_H = \left\lfloor \frac{N_H^f \times n}{N^f} \right\rfloor \tag{4.4}$$

where M_H^f and N_H^f are the free memory and CPU resources of the home node, and M^f and N^f are the free memory and CPU resources of the whole server, respectively. In short, the proportion of NUMA home node resources allocated to each I/O VM is proportional to the VM's total size.

NUIOA workload scheduling. As we observed in Section 4.2, simply configuring the VM with vNUMA is insufficient to ensure optimal I/O performance. Instead, both hypervisor and VM levels of scheduling need to be NUIOA-aware. We first identify I/O-heavy workloads running on the host; we propose three approaches for identifying these workloads, the details of which are described in Section 4.3.1. After identifying I/O-heavy workloads, we ensure that they are prioritized to run on the I/O home node whenever possible.

4.3.1 Implementation

We implemented our NUIOA-aware allocator on Ubuntu 20.04 and Debian 10 running on top of Xen 4.11. Our prototype consists of a set of userspace tools for topology and workload discovery. These tools interact directly with existing hypervisors' libraries and OS interfaces, meaning they can be easily reused across different hypervisor technology stacks, or integrated into virtualization platforms such as OpenStack.

Hypervisor-layer implementation. On the hypervisor, our allocator mainly involves detecting existing resources and their relative topologies. For example, for each device concerned by NUIOA allocation, we examine its hardware topology, including IOMMU group information as exposed by the hypervisor as well as the affinity to its home NUMA node. Next, we correlate the association of NUIOA devices to VMs, also using existing hypervisor APIs. We can directly detect Xen device passthrough by using Xen's libxl facilities, or detect VFIO-based device passthrough by probing open VFIO device files, then cross-referencing them with their IOMMU group information.

VM implementation. Upon VM boot, we first gather information about the virtual PCI topology, in order to discover the NUMA node affinities of each NUIOA passthrough

device. Following our observations in Section 4.2 that kernel-bypass solutions are most impacted by NUIOA effects both in terms of latency and throughput, we therefore focus our detection of I/O-heavy applications on detecting the presence of the aforementioned kernel-bypass platforms:

1. If a workload relies on libraries that perform direct device communication (e.g. libvma), we automatically assume this workload to be NUIOA-prone. Detection of these libraries is simply done by scanning each process's memory mapping (i.e. `/proc/[pid]/maps`)

2. We use techniques similar to hypervisor-layer NUIOA detection on I/O applications. Namely, DPDK applications can utilize kernel drivers such as UIO, VFIO or specific drivers (e.g. `mlx4-core`) for direct hardware communication; these drivers are detectable using the same technique as with the hypervisor layer.

Once we identify any NUIOA-prone workloads running on the system, we configure vNUMA on the VMs hosting these workloads, and then assign them a NUIOA affinity using OS-level tools like numactl. To maximize resource usage across all NUMA nodes while avoiding oversubscribing a single NUMA node when hosting many I/O applications, we use a *soft-affinity* scheme, where the NUIOA-prone workload is pinned to its home node when there's sufficient resources (free CPU time, free memory) to fit the workload in question. Otherwise, the workload is allowed to span multiple NUMA nodes and make use of the resources it requested.

4.3.2 Discussion

To recall, vNUMA is the main mechanism for exposing NUMA/NUIOA topology information to VMs. This topology information is prone to changes during the lifetime of the VM, either for resource utilization fairness and optimizations implemented by the hypervisor (e.g. CPU load balancing, VM live migration, memory ballooning), or by the use of certain communication mechanisms (e.g. page flipping). However, existing hypervisors and guest OSes are not designed to handle changes to their NUMA topologies, often requiring a reboot and redetection of the exposed topology.

To avoid the issue of topology changes, our approaches could be combined with those of Voron et al. [10] and Bui et al. [11] that propose dynamic vNUMA mechanisms which can be updated during VM execution. In particular, as a VM's underlying topology is changed by these operations, the hypervisor can provide the VM with topology updates that trigger reevaluation of workloads running inside the VM for NUIOA properties, and relocate them if necessary.

4.4 Evaluation

This section covers the evaluation of our NUIOA-aware allocation strategy. Our goal is to estimate the impact of our strategy on application performance. For this, we use the

benchmarks already described in Table 4.3. We examine several VM allocation schemes with regards to NUMA topologies:

1. VM fully allocated on the device's home node (our best-case, reference configuration);
2. VM fully allocated on a remote node (noted *Remote*);
3. VM split between both a home and remote node (noted *UMA*);
4. VM split between both a home and remote node, vNUMA enabled (noted *vNUMA*);
5. VM split between both a home and remote node, vNUMA enabled, with our NUIOA-aware allocation (noted *NUIOA-aware*).

We provide the VM used in our experiment with 6 vCPU, 8 GB of RAM; additionally, we allocate the privileged domain's resources entirely on the machine's node 0. If not otherwise specified, our experimental environment (software and hardware) is identical to that of Section 4.2.

Table 4.3: List of benchmarks used for NUIOA evaluation.

Benchmark (metric)	**Description**
Sockperf (latency, bandwidth)	See Section 4.2.
Perftest [88] (latency)	Generate a synthetic stream of RDMA operations: read, send, write, atomic.
Memcached [89] (transactions per second)	Remote client sending request to an in-memory key-value store.

4.4.1 Result analysis

Sockperf. Our first evaluation aims to validate our allocator strategy with Sockperf. As shown in Section 4.2, we use the benchmark to measure both network latency and bandwidth. Figure 4.7 and Figure 4.8 respectively show the latency degradation on InfiniBand and Ethernet networking; Figure 4.9 and Figure 4.10 show bandwidth figures for both interfaces. The left side of each figure shows the performance degradation on the Rx path (i.e. the VM is receiving packets); conversely, the right side of each figure corresponds to the Tx path. We note that when using the InfiniBand interface, the performance degradation *% Deg* under our NUIOA-aware allocator is lower than that of other VM allocation schemes on both the Rx and Tx paths. For instance, performance degradations reach approximately 20% at 200000 and 400000 messages per second with the vNUMA configuration, compared to below 3% under our strategy. On the other hand, workloads using Ethernet adapters with kernel I/O (Figure 4.7) show more variability and less overall performance impact. While our allocation strategy shows little improvement over other allocation schemes, it remains comparable to that of our best-case configuration, showing that our NUIOA allocator causes little

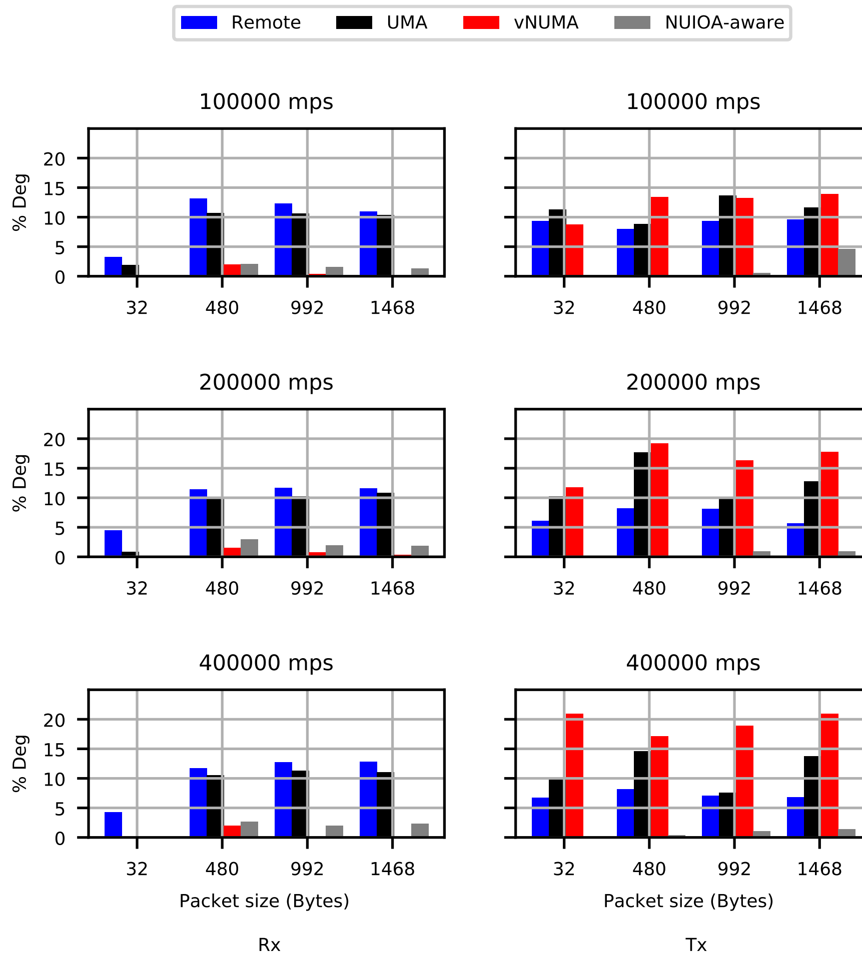

Figure 4.7: Sockperf latency evaluation on InfiniBand networking.

to no performance degradation on this particular workload. The *UMA* allocation scheme garners a performance impact on many configurations since the lack of NUMA-awareness causes remote memory accesses between the application and guest kernel; however, UMA allocation is a very common configuration for large VMs, and is in fact the default on Xen when vNUMA is not configured.

Figure 4.9 and Figure 4.10 present the bandwidth results under our NUIOA allocator compared to other allocation schemes. Similarly to our latency results presented above, we note from Figure 4.9 that our allocation strategy shows nearly zero bandwidth degradation on InfiniBand networking, while on the Ethernet device (Figure 4.10) the impact of the NUIOA-aware strategy is lower, yet our scheme remains competitive with our reference.

Perftest RDMA. Perftest [88] is a software suite for benchmarking the performance of RDMA verbs over RDMA-capable network interfaces such as InfiniBand and RoCE. We evaluate the performance of all four main operations: *send* (destination node chooses data location), *RDMA read*, *RDMA write* and *RDMA atomic fetch+add* over two operation directions: transmit (Tx) where the VM-under-test *initiates* the RDMA operation, as well as receive (Rx) where the VM-under-test *responds to* the RDMA operation. Figure 4.11 shows the results obtained from Perftest. We observe that receiving RDMA writes causes

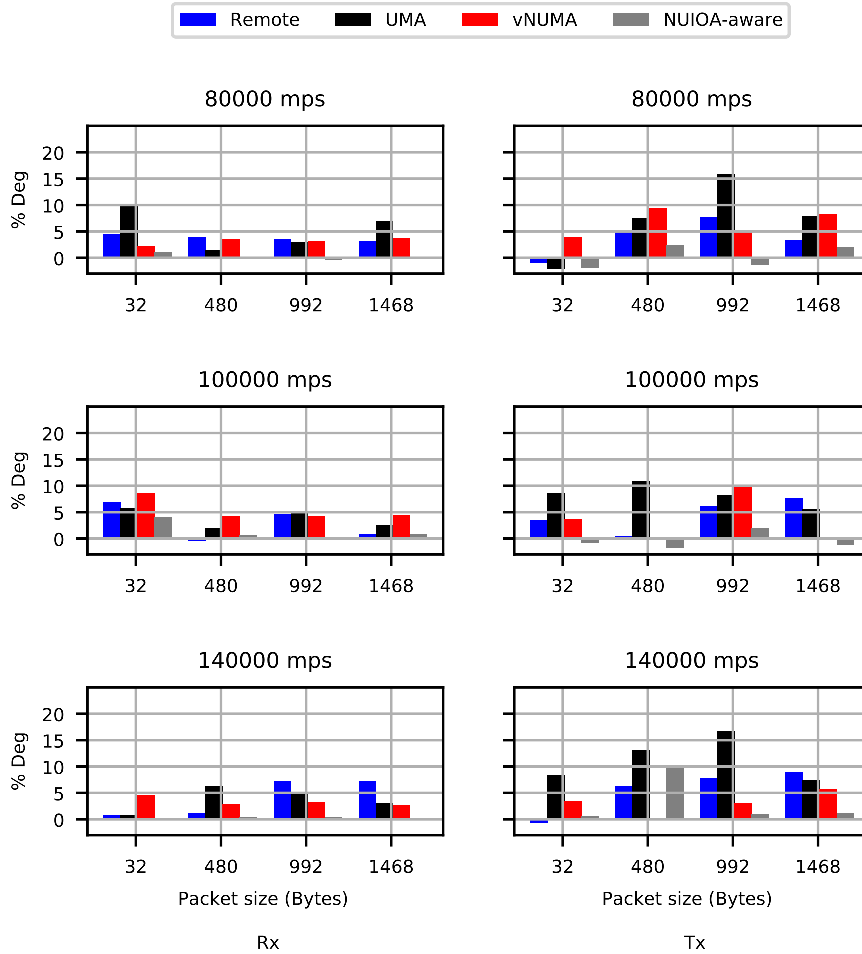

Figure 4.8: Sockperf latency evaluation on Ethernet networking.

a higher degradation than receiving RDMA reads, whereas the degradation is generally comparable when transmitting different types of RDMA operations. Nevertheless, in all cases, our NUIOA allocator shows identical performance to the reference configuration, while other configurations show anywhere from 3-10% of degradation.

FIO. We assess the impact of our allocation strategy with a file system performance benchmark named **fio**[90]. This benchmark is used to compute throughput for a realistic workload on an active disk. We deployed fio on our test VM configured with a infiniband device, and the benchmark performs IO requests to the remote machine with the data loaded into the ramdisk in order to not have the disk latency as a bottleneck. Therefore, all I/O requests use the network as it is the case with distributed file systems. The results obtained are presented in Figure 4.12. We can observe that our NUIOA-aware allocator shows almost identical performance to the reference configuration, while other configurations show anywhere from 3% to 7% of performance degradation.

Memcached benchmarks. Memcached is a popular in memory key-value store that can be used as a temporary object cache. Being an in-memory database, Memcached is mostly CPU- and memory-intensive; we therefore used the program to demonstrate that our NUIOA-aware allocation strategy can be used for all classes of applications, including CPU- and memory-heavy ones. We configure Memcached to listen over the

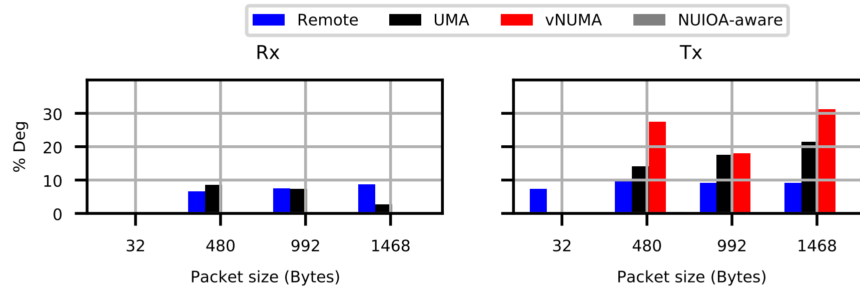

Figure 4.9: Sockperf bandwidth results on InfiniBand networking.

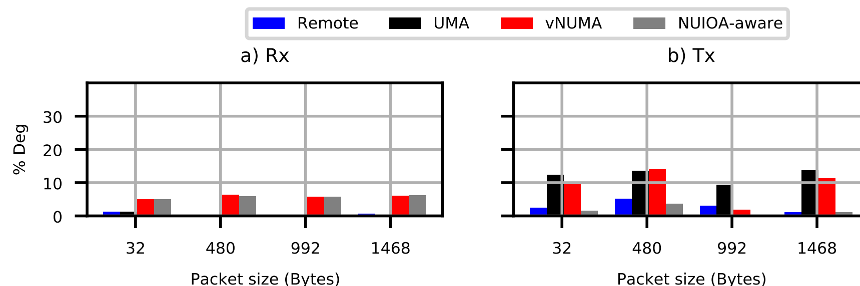

Figure 4.10: Sockperf bandwidth results on Ethernet networking.

Infiniband interface for any requests, then used the Memaslap load generator client on another physical machine to stress the Memcached server. Figure 4.13 presents the performance degradation *% Deg* for each VM configuration compared to our reference configuration. Putting Memcached on the remote node presents nearly no performance penalty; however, running Memcached on a *UMA* or *vNUMA* VM causes a staggering 30% slowdown. This implies that the penalty caused by these configurations come from NUMA remote accesses themselves rather than any NUIOA effect, and that even vNUMA is by itself insufficient to stop this overhead. Regardless, our NUMA affinity-based strategy ensures similar performance as long as Memcached fits on one NUMA node, further stressing the importance of NUMA-aware workload scheduling.

4.4.2 Scalability evaluation

In this section, we demonstrate the ability of our allocation strategy to identify I/O-heavy workloads and schedule them on their home nodes when colocated with other applications in the same VM. Using our allocation strategy, we compare the obtained results in two scenarios: a) when running Sockperf alone, and b) when colocating the Sockperf VM with another VM hosting several instances of Sysbench's CPU benchmark. Table 4.4 presents the results with each line representing an execution scenario: the first column contains the number of running instances of Sysbench; the second column presents the resulting Sockperf latency increase compared to when it is executed alone; and the last column shows the average QPS (Query Per Second) for all Sysbench instances. We observe that the performance degradation of Sockperf is close to zero in

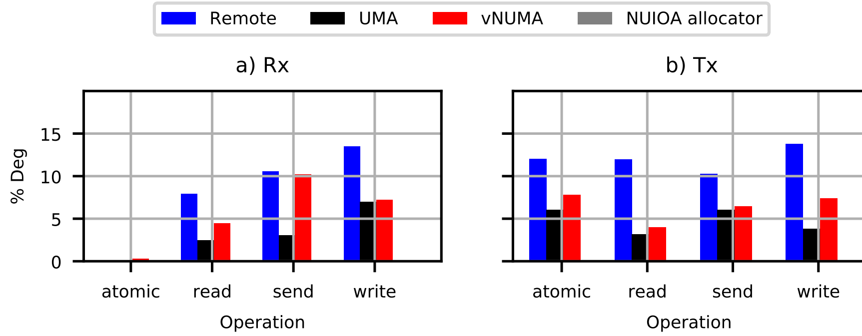

Figure 4.11: Perftest evaluation results.

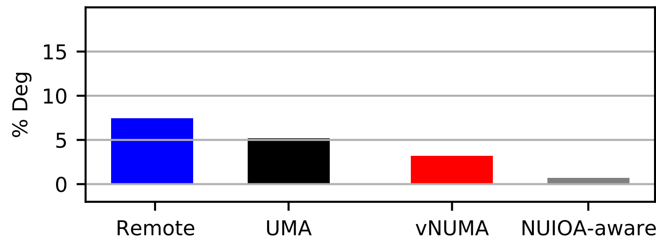

Figure 4.12: Fio evaluation results.

all scenarios, regardless of how many Sysbench instances were running. This implies that regardless of the Sysbench VM's CPU activity, our workload detector correctly identified I/O-heavy workloads running inside the VM and applied the correct affinity settings. Moreover, the performance of Sysbench is not impacted, further demonstrating that our allocation strategy does not hurt other running workloads.

Table 4.4: Performance impact on Sockperf with varying number of Sysbench instances

Sysbench instances	Sockperf slowdown (%)	Sysbench QPS
1	0.59	685.12
2	0.55	684.68
4	0.93	683.55

4.4.3 Overhead of NUIOA-aware allocation

While our NUIOA-aware allocation strategy causes minimal application overhead as seen from our evaluations, it remains that our NUIOA allocator can introduce several issues involving both performance and administration overhead aspects:

- System operators must define a VM as an I/O-heavy VM. This can be done semi-automatically using the approaches described in Section 4.3.1, or by the user simply by offering different classes of VMs (e.g. CPU-heavy, memory-heavy, I/O-heavy).
- Our soft-affinity pinning strategy only kicks in when there exists sufficient free resources on the workload's home node, deferring to the VM scheduler otherwise.

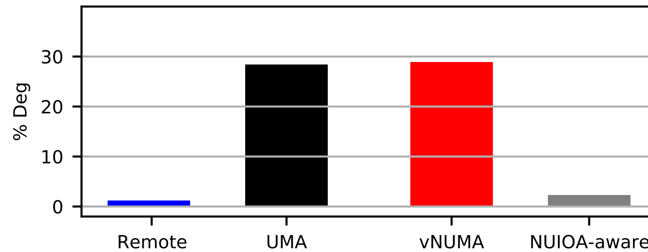

Figure 4.13: Memcached evaluation results.

Techniques such as scheduler-level soft affinity [91] can be used in the guest to prioritize scheduling I/O-extensive workloads on the home node.

- Our allocator requires enabling vNUMA on each I/O-heavy VM, which may introduce performance penalties to both I/O and non-I/O applications. However, as discussed in Section 4.3.2, our allocator can be combined with dynamic vNUMA solutions to reduce this penalty.

4.5 Related works

Several research works have focused on the problem of NUIOA management, both in virtualized and non-virtualized environments. These works can be classified into two main categories: software- and hardware-based implementations. We detail examples of both implementation categories below.

4.5.1 Software solutions

Several studies propose software solutions to manage NUIOA, mainly in non-virtualized environments. The existing state-of-the-art software solutions involve either (1) recommending users to manually pin I/O-intensive applications to their device's home node to ensure locality; or (2) implementing an automatic pinning mechanism [29, 92, 93, 94, 95], combined with migrating threads away from their home node as needed for load balancing purposes [96]. Other studies investigate the implementation of NUIOA-aware schedulers [96, 97] that extend existing NUMA-aware schedulers to include PCIe locality constraints. While these solutions can manage the NUIOA problem, as we stated in Section 4.1, some current hypervisors like Xen do not expose NUIOA locality to VMs, making these solutions inappropriate in a virtualized context.

Other works study I/O performance on NUMA architectures, particularly by investigating the resource allocation of VMs and its I/O services (e.g. privileged domains that provide I/O to other VMs) [98, 99]. These works mostly focus on colocating these privileged and user domains on the same physical NUMA nodes to ensure locality for these services; however, they do not study the problem of NUIOA in detail.

Existing systems like Red Hat Enterprise Linux or OpenStack provide recommendations for NUIOA performance optimization by allocating I/O-heavy VMs entirely on

their home node [83, 100]. While this solution ensures optimal I/O performance, it also leads to processor/memory resource contention in cases where multiple VMs share the same device, where multiple I/O devices are connected to the same node, or where one VM is connected to multiple different devices. In other words, a simple 1:1 relation between VM:device restricts the scalability of such a solution.

4.5.2 Hardware solutions

Various works propose hardware-level solutions for mitigating NUIOA effects. Works from VMware [95], Squyres [101], and Moreaud et al. [102] propose a multihomed approach by equipping each NUMA node with its separate I/O devices, therefore ensuring that applications on each node will have access to its own local devices. While effective at mitigating NUIOA effects, this approach requires significant extra investment for each class of device (extra network cards, storage devices, GPUs...), leading to higher costs, higher energy consumption and lower hardware utilization. Additionally, as pointed out by Smolyar et al. [103], installing multiple devices is not a complete solution since it cannot ensure optimal performance during thread migration, and increasing I/O performance means that modern datacenters are trending towards providing a single NIC/disk/GPU per server.

Solutions such as Mellanox SocketDirect NICs [104] and IOctopus [103] propose to avoid NUIOA effects by connecting the same device to multiple NUMA nodes through a multi-interface hardware design combined with hardware-level switching support. While promising, these solutions require the use of multi-interface devices, which comes with additional costs compared to single-slot hardware, and utilizes extra PCIe slots that could otherwise be used with other devices.

4.5.3 Positioning of our work

Our work focuses on the problem of NUIOA scheduling, which has yet to receive widespread attention compared to (processor and memory-level) NUMA scheduling. In contrast to other works on NUIOA, we propose an unique scheme for optimizing resource allocation of I/O workloads that reduces the effect of NUIOA without requiring hardware modifications, while preserving the scalability benefits of NUMA architectures namely by not requiring I/O workloads to be wholly allocated on a single NUMA home node.

4.6 Summary

In this chapter, we studied in detail the various effects of NUIOA on I/O performance using multiple different workload and device types.

We thoroughly evaluated the impact of NUIOA on application performance in VMs and showed that current systems still experience performance impacts caused by NUIOA, with up to a 20% increase in latency and 10% decrease in bandwidth. We provide recommendations to resolve this issue.

We proposed a NUIOA-aware allocation strategy that distributes VMs over multiple physical NUMA nodes while informing VMs of NUIOA affinities, and showed that our scheme improves I/O performance similar to an optimal behavior, therefore preventing the 20% impact on various operations compared to the default VM allocation strategies.

This chapter was originally published as: Tu Dinh Ngoc, Boris Teabe, Daniel Hagimont, and Georges Da Costa. Optimized resource allocation on virtualized non-uniform I/O architectures. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 432–441. IEEE, 2022. doi:10.1109/CCGrid54584.2022.00053. © 2022 IEEE. Reprinted under permission.

Chapter 5

NVMetro: Enhancing mediated NVMe virtualization with eBPF

Contents

5.1 Overview **66**

5.2 Design of NVMetro **67**

5.2.1 General overview 67

5.2.2 NVMetro's design criteria in detail 69

5.2.3 I/O router and classifier 70

5.2.4 Userspace I/O functions 71

5.3 Use cases **72**

5.3.1 Transparent data encryption 72

5.3.2 Live disk replication 76

5.3.3 Implementation effort 76

5.4 Evaluation **77**

5.4.1 Experimental setup 77

5.4.2 Basic performance evaluations 78

5.4.3 Disk encryption evaluations 81

5.4.4 Disk replication evaluations 82

5.4.5 Overhead evaluations 83

5.4.6 NVMetro's flexibility and ease of use in perspective 84

5.5 Related works **86**

5.6 Summary **88**

5.1 Overview

Virtualization of VM storage is often done in the form of *storage functions* covering certain use cases, e.g. data encryption, compression, replication, etc. Most of these technologies take one of two forms: a hypervisor-based, fully-featured stack that makes extensive use of OS features (e.g. QEMU's own virtual disk implementation, Linux's in-kernel Vhost), or a hardware-based stack that forgoes OS-level management in return for improved performance (e.g. device passthrough, SPDK [105]).

Hardware-based stacks deliver the most storage performance to VMs, but have the drawbacks of reduced manageability, difficulty of use and a limited feature set. For instance, device passthrough-based frameworks such as SPDK require assigning an entire device to its userspace driver. As a result, the device cannot be accessed via the kernel API; disk access must be done with SPDK-specific APIs, or through a compatibility bridge (e.g. the DPDK kernel-native interface [106] for networking or FUSE [107] for virtual filesystems). Userspace solutions like FUSE also come with issues that can severely degrade performance [108]. Alternatively, single-root I/O virtualization (SR-IOV) can be used to partition one physical device into a set of PCIe virtual functions that can be independently shared to each VM; however, SR-IOV is restricted to a single use case of isolation between VMs, and gives the host next to no control or visibility over how each VM uses its resource partition, as the guest NVMe driver bypasses the host hypervisor to directly communicate with hardware.

All in all, these drawbacks are reasons to choose in-kernel I/O implementations over a higher-performing userspace or hardware-based one [109]. Yet OS-based stacks, while being easier to use, struggle to keep up with hardware. At the level of I/O performance demonstrated by the Intel Optane P5800X cited above, software becomes a significant part of I/O overhead, with kernel code taking nearly half of the time cost of a read/write system call [13]. Moreover, the implementation of complex storage functions can be challenging due to a lack of tooling integration in the kernel. Consider an example of a storage function that performs data encryption using Intel SGX. Such an application can be easily written with Intel's existing SDK; however, implementing said function inside the kernel requires a new, kernel-specific SDK, which is a considerably more challenging task. To summarize, current solutions lack the flexibility for implementing complex storage software; therefore, *we would need a more scalable and adaptable solution that meets all the challenges of virtual storage.*

In this chapter, we present *NVMetro*, a solution for efficiently managing storage in virtual machines. With NVMetro, we aim to ease the creation of flexible storage logic *without sacrificing either strong performance or security*. NVMetro proposes an unique solution that offers multiple I/O paths for handling virtual storage requests: (1) a *fast path* adjacent to hardware NVMe devices; (2) a *kernel path* attached to the host's kernel storage stack; and finally (3) a *notify path* controlled by an userspace I/O function. These I/O paths are controlled by a central *I/O router* accompanied with interfaces for specifying policies that choose the best path for each individual request, as well as a support framework for userspace applications using the notify path.

Our design of NVMetro is guided by five main criteria:

- **Flexibility**: the **key ability** of NVMetro to provide fine-grained partitioning and control of storage requests, thus letting it adapt to multiple types of storage functions;
- **Performance**: ensuring that NVMetro does not significantly degrade I/O performance compared to other solutions;
- **Isolation**: making sure that NVMetro does not break the security model of storage virtualization;
- **Compatibility**: ensuring that NVMetro works with all VMs supporting the NVMe specification;
- **Ease of use**: creation of a storage framework that minimizes the development effort needed to write a storage function with NVMetro compared to existing solutions.

To accomplish our design criteria, NVMetro uses two main components: (1) an *I/O router* supporting *pluggable classifiers* based on Linux's *Extended Berkeley Packet Filter* (eBPF) for encoding custom logic into the storage virtualization pipeline; and (2) a kernel-user API that assists the creation of *userspace I/O functions* (UIFs) for high-performance storage processing.

In Section 5.2, we explain the goals and design criteria of NVMetro, show their applicability to a range of storage function use cases, and make comparisons to the designs of other works. Following these criteria, we present the main design of NVMetro's I/O routing, classification and UIF components. Section 5.3 investigates in depth two NVMetro use cases: a data encryption function and a data replication function. We show in Section 5.4 the performance of NVMetro and its use cases under various workloads. Our results demonstrate that NVMetro takes good advantage of the storage performance of modern hardware, with our NVMetro-based disk encryption for VMs being up to 3.7× faster than an in-kernel virtual encrypted disk based on `dm-crypt+vhost-scsi` while using as little as 0.9× the CPU during heavy loads. Section 5.5 gives an overview of other storage virtualization and computational storage approaches, and Section 5.6 concludes this chapter.

5.2 Design of NVMetro

In the following sections, we give a general overview of our solution, then describe each design criterion in further depth.

5.2.1 General overview

NVMetro aims to ease the development of fast and flexible storage functions for VMs. We continue from our observation in Section 5.1 that current storage virtualization solutions only provide one possible access method; they are effectively "all-or-nothing" in the sense that once a storage function developer selects a specific virtualization API, they

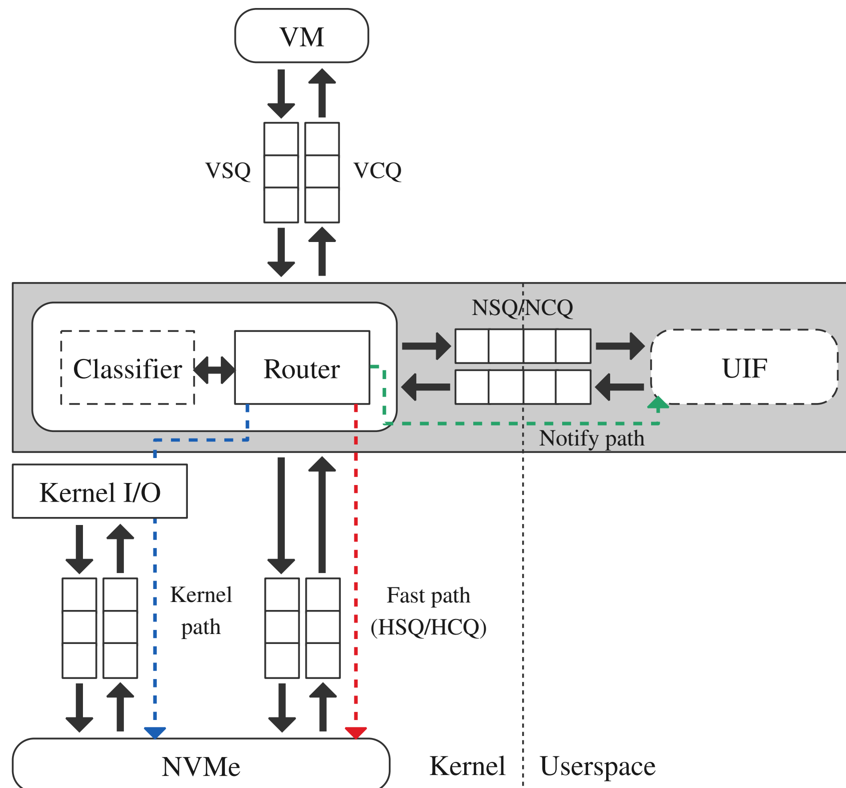

Figure 5.1: NVMetro architecture and I/O paths. Customizable components are drawn in dashed outline. VSQ/VCQ, HSQ/HCQ and NSQ/NCQ denote *virtual*, *host* and *notify* submission and completion queues (see Section 5.2.3).

cannot easily switch to another API to meet their new requirements. In NVMetro, we give developers multiple ways to process I/O requests with various trade-offs between performance, flexibility and ease-of-use depending on their use case.

Our solution operates in the hypervisor, and presents itself as a virtual NVMe controller in each concerned VM, intercepting and servicing I/O requests from the VM. This is done in accordance with the NVMe protocol, i.e. NVMetro appears as a normal NVMe device inside the VM. As a consequence, all VMs equipped with a NVMe driver are compatible with NVMetro by default without needing guest-side modifications. Virtual controllers can be attached to an entire NVMe namespace on the drive, or a fixed partition as defined by a partition table (e.g. GPT table). NVMetro also supports creating multiple queue pairs, preserving NVMe's parallelism benefits.

Figure 5.1 summarizes NVMetro's main components. In short, requests coming from the VM pass through an *I/O router*. This router is informed by a customizable *I/O classifier* to route requests through one of three *I/O paths*: (1) a *fast path* to a physical NVMe device (red arrow); (2) a *kernel path* (in blue); and (3) a *notify path* (in green) to an external UIF.

The I/O router needs to inspect incoming requests to find the most appropriate I/O path. As the selection of which path to use must depend on each particular I/O function's use case, this step is done through *I/O classifiers* provided by the storage function developer. To ensure that the classifiers determine a request's I/O path as quickly as possible, their custom code runs directly inside the host kernel inside an

isolated environment.

Next, we describe the tradeoffs to each I/O path that the classifier must take into account in its selection logic. The fast path is the simplest one described in our system, and involves sending each request directly to an underlying NVMe drive for processing. As a result, it is the most performant I/O path in NVMetro for most requests.

The kernel I/O path translates I/O requests and sends them through the host kernel's block device architecture. This path incurs a request translation cost, and is only usable with requests that follow the Linux kernel's storage semantics (versus NVMe-specific or vendor-specific commands); however, it is compatible with Linux's block layer features (e.g. device mapper), as well as with non-NVMe-based backing devices.

NVMetro's *notify path* exports requests for processing outside of the host kernel. Said request is then handled by an *userspace I/O function* (UIF) that makes up part of the desired storage function. These UIFs are used in cases where a) in-kernel request processing architecture is insufficient; or b) extra isolation of the storage function is required. To ease the creation of these UIFs, NVMetro includes a C++-based framework that takes care of basic UIF housekeeping tasks.

Unique to NVMetro is that our I/O classifiers can run multiple times for each request, and the output of each execution dictates the next destination to which it should be sent. This feature aims to assist complex use cases where a request needs multiple processing stages before it could be completed. This routing can repeat forming a state machine where the classifier models each request's transition between several states until it is completely fulfilled. Rather than filtering at every level of the I/O stack, the classifier is only invoked at key decision points during its lifetime, thus saving CPU and memory usage.

5.2.2 NVMetro's design criteria in detail

Flexibility. In existing storage stacks (MDev-NVMe [110], Linux's device mapper, FreeBSD's GEOM, SPDK, etc.), functionalities such as encryption, quality-of-service, etc. needed in each stack needs to be implemented in the stack itself. In contrast, NVMetro provides a more fine-grained storage filtering, integrating UIFs into the I/O request path as desired. In complex use cases such as encrypted key-value stores, NVMetro eases the integration of relevant technologies (e.g. Intel SGX as presented above) without affecting unrelated I/O requests thanks to our isolated classifier architecture.

In NVMetro, UIFs can be combined (1) generically, by programming the I/O classifier to forward requests between UIFs; (2) by direct IPC between UIFs; or (3) by combining all function logics into a single UIF. Moreover, our modular design lets storage administrators install, migrate and remove storage functions on the fly, a desirable feature for avoiding VM reboots needed in some solutions (e.g. `vhost-scsi`).

Performance. NVMetro adds routing on top of a storage virtualization system (MDev-NVMe); therefore, it necessarily imposes some overhead over this existing solution. However, our key contribution comes from shortcut processing of I/O requests, running a custom classifier followed by redirecting to the next hop as quickly as possible. In

other words, commands that can be served directly by the physical disk are immediately sent there for processing; only those classified as requiring extra processing will be sent to an alternative I/O path. NVMetro maintains the benefit of mediating I/O requests without introducing a significant performance impact.

Isolation. In-kernel storage virtualization (Vhost, MDev-NVMe) keeps functional logic (e.g. encryption, replication, caching...) inside the kernel for performance reasons. This decision also increases the attack surface of these solutions. In NVMetro, we offload most of the work to isolated subsystems such as sandboxed classifiers and userspace processes. The remaining I/O router components in the kernel only minimally processes incoming requests, thus reducing our attack surface.

Compatibility. The large software stack in solutions like Vhost complexifies the implementation of certain special I/O commands (e.g. block unmapping, security commands). Lack of support for any such command at any storage layer effectively prevents it from being used by the guest. Not only is NVMetro compatible with the base NVMe specs, commands deemed safe by the I/O classifier can be passed directly to hardware, enabling the use of vendor extensions for performance or security purposes. NVMetro can also easily adapt to any new NVMe features (e.g. the KV command set) by adjusting the classifier without changing the host kernel.

Ease of use. Kernel-bypass solutions like SPDK and `vfio-user` provide high performance through userspace polling drivers. However, these low-level drivers require significant reengineering, take up exclusive control over the device, and cannot use the diagnostic features already built into the Linux kernel. This is, among other reasons, why Cloudflare chose to use the Linux kernel's TCP networking stack rather than DPDK [109]. In NVMetro, each UIF chooses the programming languages, libraries and APIs that best suit its purposes. We demonstrate this property of NVMetro by implementing a data encryption UIF based on the Intel SGX SDK.

Following the design criteria discussed above, the next sections detail NVMetro's two core components: the *I/O router and classifier*, and its accompanying *userspace I/O functions*.

5.2.3 I/O router and classifier

NVMetro implements a set of data paths for the processing of I/O requests as soon as they are received by the host. We adapt the device queue shadowing method in MDev-NVMe [110]: NVMetro's I/O router receives commands from the guest using *virtual submission queues* (VSQ), and sends results to the guest using *virtual completion queues* (VCQ) (see Figure 5.1).

To accomplish the goal of injecting custom logic into the kernel without compromising its security, we selected eBPF as the platform of choice for our I/O classifiers. I/O classifiers (also called eBPF classifiers) modify the request in two complementary fashions. The first step is *direct mediation*, where the classifier directly modifies a

request command's content. With direct mediation, each eBPF classifier can limit the privileges of each command, e.g. by translating its requested logical block address (LBA) to the underlying device's real LBA (cf. MDev-NVMe which implements LBA translation directly inside its kernel module). The second step is *request routing*. With request routing, requests are *routed* by NVMetro or stopped by sending an error status to the VM's VCQ.

NVMetro implements an iterative routing approach, i.e. a request can traverse multiple hops following the classifier's policy. Iterative routing is managed by a routing table that keeps track of each request's state at classification time. Following direct mediation, requests can be forwarded to the queue types corresponding to the I/O paths shown in Figure 5.1: (1) the fast path, which redirects requests to the underlying physical device's I/O queues, called the *host submission queues* (HSQ) and *host completion queues* (HCQ); (2) the kernel I/O path, which sends requests through Linux's block device subsystem; or (3) the notify path, which links to a UIF through *notify submission/completion queues* (NSQ/NCQ). Along with the VSQ, the CQs of these paths are actively polled by host worker threads belonging to NVMetro's router component. These worker threads are shared between multiple VMs in a round-robin fashion; each VM is individually tracked such that polling on said VM is disabled in case of inactivity.

When sending requests between components, NVMetro minimizes unnecessary memory copies even under long request paths. NVMetro only passes around each request's 64-byte command block, while the scatter-gather lists and data pages stay inside the VM's memory. Storage functions that need to alter data pages (e.g. encryption) can be optimized with the NVMe Host Memory Buffer (HMB) feature, where altered pages are stored inside the HMB region inside the VM's memory and passed directly to the physical device.

The I/O classifier can also send one request to multiple targets at the same time if necessary. This is useful when the device and UIF need to work at the same time, e.g. during snapshotting, backup or mirroring. Moreover, the classifier can install additional *hooks* into the request. Hooks define certain events that happen during the request's lifecycle, e.g. when a request has been processed by the hardware or UIF. Each hook calls its I/O classifier again, deciding the next course of action until the request is satisfied. To prevent infinite classifier loops, a request's TTL is decremented every time the classifier runs, and requests with an expired TTL are rejected.

5.2.4 Userspace I/O functions

UIFs are normal programs that process each command coming through the notify path according to the storage function's requirements. Each UIF opens NSQs/NCQs as file descriptors, and maps them into its address space through `mmap()` calls. It then poll its NSQs for requests coming from the I/O router. It also has access to the VM's memory to read and write request data as needed. Our UIFs use an *adaptive polling* approach, where they can switch between active polling and OS-assisted waiting (using `poll()/epoll()`) depending on the activity level. This approach also permits serving multiple VMs using multiple UIFs in the same process to lower the CPU cost of busy

polling. When the UIF finishes its processing, it returns a status code to the kernel via the NCQ.

To reiterate Section 5.2.1, UIFs hold a critical role in handling requests that should be further isolated, or cannot be easily implemented inside eBPF classifiers. For instance, as stated in Section 2.5, eBPF programs run under multiple restrictions to ensure the kernel's integrity. Additionally, the time-critical, in-kernel nature of eBPF classifiers makes some features difficult to utilize (e.g. timers, memory allocations).

As stated above, UIFs are free to choose the best APIs for fulfilling requests they receive. In other words, they can use basic `read()` and `write()` calls to serve data from a backend file, use `io_uring` to improve performance, or even send HTTP requests to a cloud service. However, to help the creation of UIFs, we created an **UIF framework** to provide NVMetro UIFs with the following services:

1. Setting up notify queues and `io_uring` mappings for communication with the NVMetro router;
2. Configuring polling threads for I/O queues;
3. Parsing of incoming NVMe commands, as well as reading and writing of data pages from the VM;
4. Exposure of requests from the VMs as UIF events.

Our framework spans only 1100 lines of C++, and helps creating UIFs with minimal programming effort. We provide an example of a UIF under our framework in the next section.

5.3 Use cases

In the following sections, we detail two examples of storage virtualization functions based on our NVMetro framework: a function that encrypts data transparently on disk with secure enclave integration, and a function that replicates data between two disks. For each storage function, we present its general request lifecycle, followed by describing the roles of its components, namely the I/O classifier and related UIF.

NVMetro storage functions (classifier + associated UIFs) are managed by the virtualized cloud system operator. As stated above, our solution is exposed as a virtual NVMe controller inside each VM, with an additional control interface on the host. System administrators attach each virtual controller to a namespace or partition on a backend NVMe device. These control interfaces can then be used to insert eBPF classifiers and attach UIFs for functions they wish to apply to each VM.

5.3.1 Transparent data encryption

We created a storage function for the encryption-at-rest of a virtual disk, a critical feature in cloud environments for the protection of sensitive data. Figure 5.2 shows the general lifecycle flowchart of an I/O request under our disk encryption function. When

NVMetro intercepts a request, our eBPF classifier categorizes its type as either a read or write request. Read requests are sent to the device to retrieve the ciphertext; once this step finishes, the command is routed over to a UIF for decryption before finishing. Write requests are routed directly to the UIF, which encrypts the to-be-written data and writes it to the physical device.

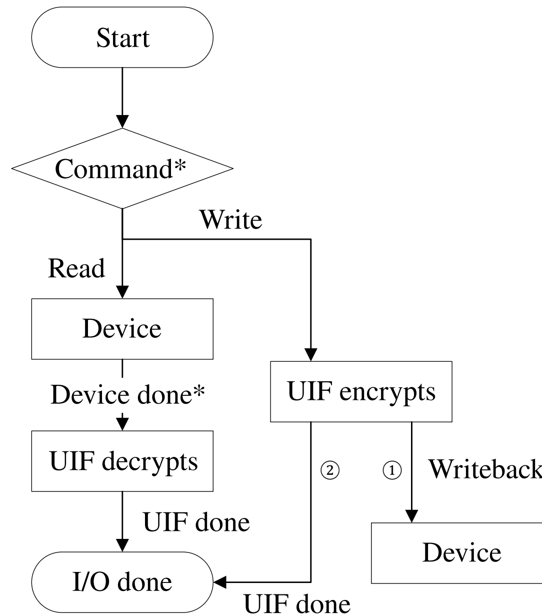

Figure 5.2: Lifecycle of an I/O request with data encryption. The asterisk (*) denotes classifier invocation points.

I/O classifier. To implement data encryption, we specified two rules in our I/O classifier: (1) with read commands: send the command to the physical disk, then once the disk read completes, forward the request to the UIF for decryption; (2) with write commands: send the command to the UIF for encryption, and forward it to the disk afterwards for writing. Our classifier runs at two critical decision points: once at the beginning of the request pipeline, and once more during a read command when the device finishes its read.

Listing 5.1 goes into detail on the implementation of our encryption I/O classifier. The function `encryptor_classify` is our classifier's entry point, and is called every time the classifier is needed (see Figure 5.2). Each classifier is given an I/O context `ctx` that contains information about the current request. Depending on the request's processing stage (lines 15-26), the classifier must decide the next course of action:

- Upon a new request (`HOOK_VSQ`), the classifier reads the command's type `opcode` (line 15). For read commands, the classifier instructs the router to send this request to the device (`SEND_HQ`), and to invoke the classifier again when the device responds (line 18). For write commands, the request is sent through the notify path (`SEND_NQ`). Once the UIF responds, the router immediately finishes the request without having to call the classifier again (`WILL_COMPLETE_NQ`).

- When the previously-mentioned read finishes (HOOK_HCQ), the classifier checks the device's read error code. If an error occurred, this error is immediately forwarded to the VM (line 9); otherwise, the read continues in the UIF (line 11).

Our example demonstrates both types of request modification available to a classifier:

- Direct mediation: by returning a NVMe status code (e.g. line 9, which forwards the physical device's status code). This status code is sent to the VM to stop the request;

- Request routing: The classifier chooses the target I/O paths to route our request. It can install a new hook (line 18) or automatically complete the request when its targets finish processing (lines 22 and 25).

Listing 5.1: Encryption eBPF classifier code.

```
1  int encryptor_classify(struct ctx *ctx) {
2    switch (ctx->current_hook) {
3    case HOOK_VSQ:
4      /* new request */
5      return encryptor_begin(ctx);
6    case HOOK_HCQ:
7      /* read device done, check for error */
8      if (ctx->error)
9        return ctx->error | COMPLETE;
10     else
11       return SEND_NQ | WILL_COMPLETE_NQ;
12   }
13 }
14 int encryptor_begin(struct ctx *ctx) {
15   switch (ctx->cmd.common.opcode) {
16   case nvme_cmd_read:
17     /* read commands that need reading ciphertext */
18     return SEND_HQ | HOOK_HCQ | WAIT_FOR_HOOK;
19   case nvme_cmd_write:
20     /* write commands that need encrypting,
21      * UIF will finish the command */
22     return SEND_NQ | WILL_COMPLETE_NQ;
23   default:
24     /* send to device */
25     return SEND_HQ | WILL_COMPLETE_HQ;
26   }
27 }
```

Userspace I/O function. Our encryption UIF performs three tasks: (1) in-place decrypting of ciphertext from the physical device; (2) encrypting of plaintext from the

guest into a temporary buffer; and (3) writing of ciphertext from step (2) to disk with io_uring.[5] Our encryptors use the standard XTS-AES algorithm and are compatible with Linux's dm-crypt.

Listing 5.2: Request processing code of encryption UIF.

```
1 bool uif::work(nvme_cmd &cmd, u32 tag, u16 &status) {
2   switch (cmd.common.opcode) {
3   case nvme_cmd_read:
4     status = do_read(cmd);
5     return false; /* synchronous response with status */
6   case nvme_cmd_write:
7     do_write_async(cmd, tag);
8     return true; /* asynchronous response */
9   }
10 }
11 u16 uif::do_read(nvme_cmd &cmd) {
12   for (auto data = parse(cmd); !data.at_end(); data++)
13     if (!decrypt(/*inplace*/ *data, data.lba()))
14       throw std::runtime_error("cannot decrypt");
15   return NVME_SC_SUCCESS;
16 }
17 void uif::do_write_async(nvme_cmd &cmd, u32 tag) {
18   auto data = parse(cmd);
19   auto ticket = new iovec_ticket({.tag = tag});
20   auto buf = malloc(data.nbytes()); /* temporary buffer
      */
21   /* encrypt data into temporary buffer */
22   for (; !data.at_end(); data++) {
23     auto block = buf.subspan(data.block_offset(), data.
         lba_size());
24     if (!encrypt(/*out*/ block, /*in*/ *data, data.lba())
         )
25       throw std::runtime_error("cannot encrypt");
26   }
27   /* write to disk from the UIF with io_uring */
28   ticket->iovecs.push_back({buf, data.nbytes()});
29   queue_writev(ticket, data.disk_addr());
30 }
```

Listing 5.2 shows an abbreviated version of our UIF code. Each UIF is represented by a C++ class (uif) following our implementation interface. Our framework passes incoming requests to the UIF's work function, which again classifies the request's type (lines 2-9). In case of a read, the implementation is straightforward: the UIF iterates over the data blocks coming from the device (line 12), then immediately decrypts them

[5]Note that the HMB feature can speed up encryption (see Section 5.2.4).

in-place (line 13) and informs the VM of decryption success (line 15). In case of a write, the UIF allocates a temporary buffer (line 20) which is used to encrypt data to be written (lines 22-26). The temporary buffer is written to disk with `io_uring` (lines 28-29) and the request completes when this write finishes. As seen from the code snippet, our UIF framework takes care of queue handling, request and memory management, while the UIF code only needs to encrypt and decrypt data. Moreover, our UIF framework supports all C++ features and libraries, making the UIF development environment easier than that of e.g. a Linux kernel module.

We implemented two encryption UIFs using the same I/O classifier: one normal UIF, and one running inside Intel SGX. Both versions of the enclave use the AES-NI instructions to accelerate encryption operations, the same instructions as used in `dm-crypt`, SPDK and other disk encryption software. Our SGX-based UIF stores the cryptographic key inside a hardware enclave. Both UIFs share substantial amounts of code, with the SGX version needing only $\approx$ 120 lines of code for the encryption enclave.

5.3.2 Live disk replication

We created a mirroring UIF that replicates data between two NVMe drives: a primary drive attached directly to the local host, and a secondary drive attached to a remote host. The two hosts are connected together using NVMe over Infiniband.

Our I/O request pipeline is implemented as follows: our classifier passes read requests directly from the guest to the primary disk, while write requests are sent to both the primary disk and the UIF. The UIF then forwards the write request to the secondary disk using `io_uring`. The mirroring process is synchronous, meaning that writes are not completed until both the local and remote disks finish servicing the request; this allows easy reusing of the VM's data buffers.

5.3.3 Implementation effort

As stated in Section 5.2.4, our UIF framework facilitates the implementation of fast and simple storage UIFs. Table 5.1 shows a detailed breakdown of the number of lines of code needed for each storage function that we implemented. Note that our normal and SGX encryptor functions share the same classifier and 80% of UIF code.

Table 5.1: Source code sizes of NVMetro classifier and UIF implementations.

Function	**Component**	**Lines of code**
Encryptor	Classifier	32
Encryptor	Normal UIF	520
Encryptor	SGX UIF + enclave	501
Replicator	Classifier	16
Replicator	UIF	307
Framework	—	1116

5.4 Evaluation

Our goal for the evaluation of NVMetro is twofold:

1. Compare the I/O performance of NVMetro to existing solutions in the basic use case;

2. Show our UIF framework's flexibility and ease of use through various real-world storage function use cases.

5.4.1 Experimental setup

We evaluate the performance of NVMetro using our both UIFs presented in Section 5.3 with multiple different workloads. These workloads are categorized into two groups: firstly, benchmarks of I/O performance under various configurations with `fio` [90]; and secondly, database evaluations using the YCSB suite [111].

We use two platforms for our evaluations: two Dell PowerEdge R420 servers, each equipped with 2x Intel Xeon E5-2420 v2 and 48 GB of RAM for most evaluations; and a Dell Precision 7540 laptop with an Intel Core i5-9400H and 16 GB of RAM for disk encryption evaluations with Intel SGX. Each machine is equipped with a Samsung 970 EVO Plus 1TB SSD for evaluation purposes. Experiments are conducted inside a QEMU VM with 6 GB of RAM and 4 physical cores (servers)/2 physical cores (laptops) running Ubuntu 20.04.

`fio` evaluation setup. To evaluate the raw performance of NVMetro, we executed `fio` while varying the I/O block sizes, benchmark modes (random, sequential, read/write/ mixed), queue depths (QD), and number of parallel jobs. We ran each experiment 3 times, and recorded the resulting average I/O per second (IOPS). We measured the whole-system CPU consumption of each experiment to compare the solutions' performance impacts. Table 5.2 shows a detailed list of configurations.

Table 5.2: List of `fio` benchmark configurations.

Block size	Mode	QD	Nr. jobs
512	Random read (RR)	1, 128	1
512	Random write (RW)	1, 128	1
512	Mixed random R/W (RRW)	1, 128	1
512	Random read (RR)	128	4
512	Random write (RW)	128	4
512	Mixed random R/W (RRW)	128	4
16K	Sequential read (SR)	1, 128	1, 4
16K	Sequential write (SW)	1, 128	1, 4
16K	Mixed sequential R/W (SRW)	1, 128	1, 4
128K	Sequential read (SR)	1, 128	1, 4
128K	Sequential write (SW)	1, 128	1, 4
128K	Mixed sequential R/W (SRW)	1, 128	1, 4

Additionally, we performed latency evaluations of various storage solutions with fio. Each solution is tested under a fixed I/O rate of 10,000 operations per second, while varying the block sizes and queue depths, and the resulting median and 99th-percentile latencies are reported for each configuration.

YCSB evaluation setup. We benchmarked NVMetro using the YCSB benchmark version ce3eb9c along with its 6 built-in workloads. We configured each workload to run on RocksDB/ext4 filesystem; to minimize FS overhead, we turn off the journal, discards and access time features. Each workload had 1 million operations and was run 3 times on a dataset of 3 million records. We present two evaluation scenarios: one with a single YCSB job and database instance, and another with four parallel YCSB jobs, each with its own DB instance.

5.4.2 Basic performance evaluations

In this section, we are interested in measuring the overhead of NVMetro compared to other storage solutions: direct PCIe passthrough; MDev-NVMe (implemented by Maxim Levitsky [112]); paravirtualized disk with in-kernel vhost-scsi; virtual disk using QEMU's virtio-blk with io_uring; and finally, SPDK's vhost-user-based virtio-blk. NVMetro uses a dummy eBPF classifier without UIF.

Figure 5.3 shows the performance of NVMetro compared to the storage solutions presented above in the fio benchmark. We observe that in all configurations, NVMetro with a dummy eBPF classifier performs similarly to MDev-NVMe and SPDK. Similarly, NVMetro reaches comparable performance to that of direct passthrough to the underlying VM. Being userspace-based without kernel bypass, QEMU's virtio-blk performs significantly worse than NVMetro at higher I/O rates and lower queue depths; for example, 512B random read performance on NVMetro is 2.7× faster than QEMU at QD1/1 job. QEMU regains performance at higher QDs, potentially due to its ability to redistribute I/O requests across multiple worker threads; in fact, QEMU at 16K/QD128/1 job performs the best overall, being between 19% to 32% faster than NVMetro. In comparison, vhost-scsi despite being completely in-kernel falls behind in performance, being one of the worst performers in fio regardless of the benchmark configuration.

Figure 5.4 shows the median and 99th-percentile request latencies under various fio configurations, the bar height represent the median while the 99th-percentile is represented by the whiskers' height. Among our tested configurations, a pattern emerges where NVMetro, MDev-NVMe and SPDK, being polling-based, share approximately the same median and tail latencies. Direct PCIe passthrough without polling falls behind with a median latency 18.2% higher than NVMetro at 512B RR and 9.1% higher at 512B RW, potentially due to the overhead of forwarding device interrupts to the guest. Vhost, despite being purely in-kernel, exhibits poor latencies even at our low I/O rate, namely 73.6% higher at 512B RR and 97.6% higher at 512B RW. QEMU's userspace-based storage virtualization again performs even worse, with 3.4× higher median random read latency and 4.1× higher write latency at 512B. Concerning tail

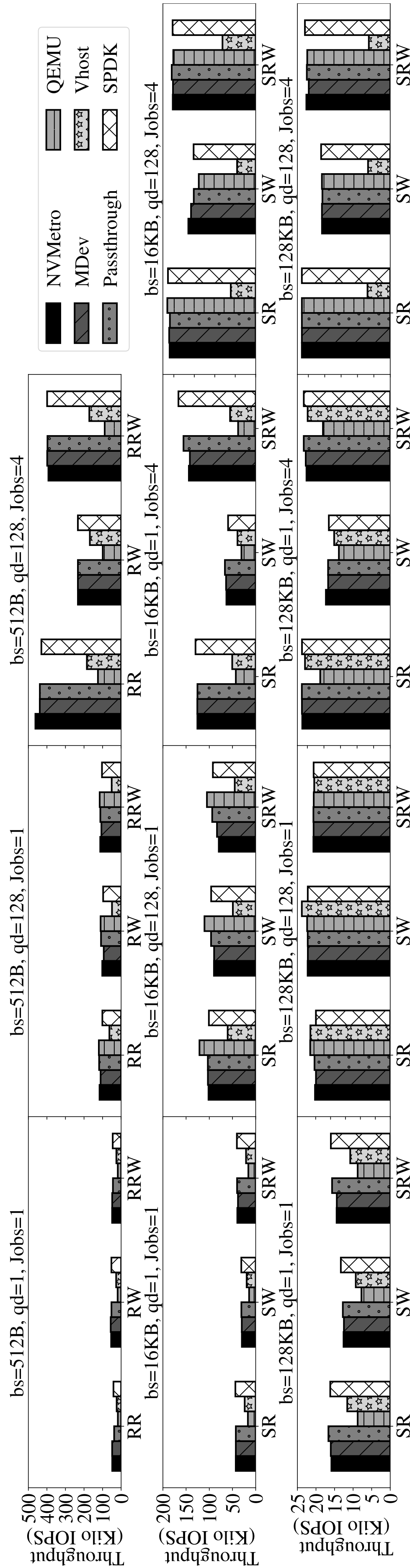

Figure 5.3: Basic evaluations: `fio` performance for each workload configuration and storage virtualization method.

latencies, the only solution with a lower 99th-percentile write latency than NVMetro is SPDK, at 5.9%, 18.0% and 13.0% for 512B, 16K and 128K blocks.

Figure 5.5 shows the scaling behavior of NVMetro with an increasing number of small VMs. Each VM in this experiment is equipped with 2 GB of RAM, 1 dedicated physical core, and a dedicated partition on the shared NVMe namespace.[6] We set up NVMetro to use one host kernel worker thread to concurrently serve all VMs. All evaluations were performed at a block size of 512B. We observe that the system throughput increases gradually as we add more VMs, confirming our solution's scalability even when colocating multiple VMs.

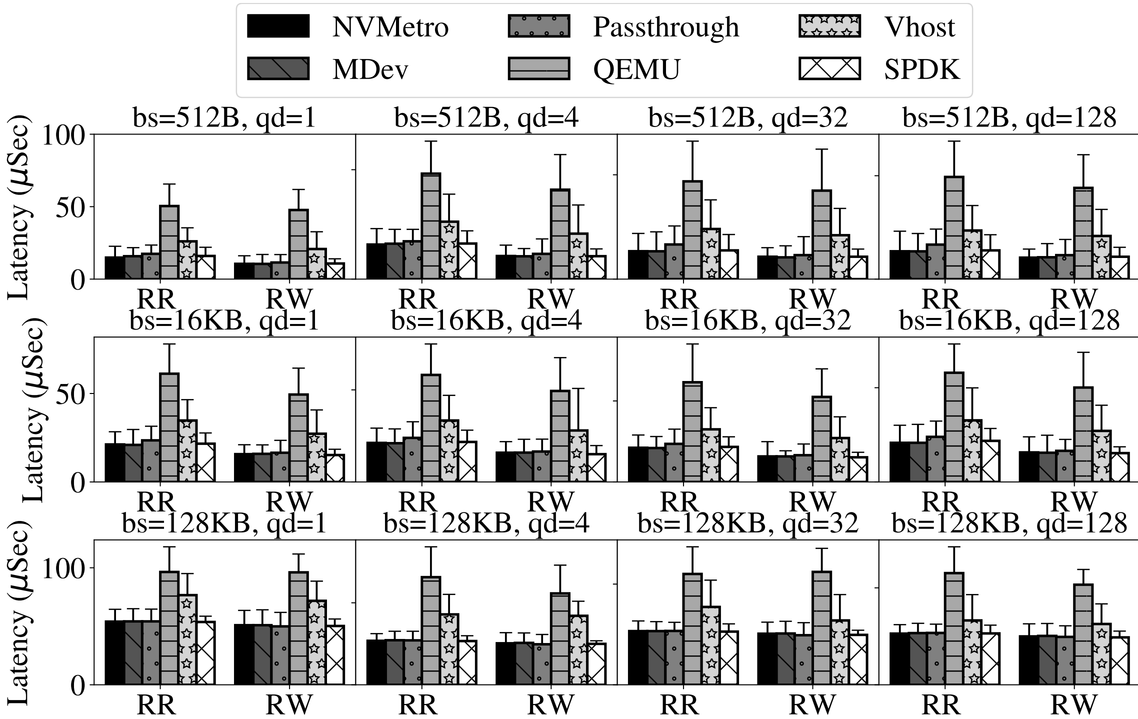

Figure 5.4: NVMetro latency evaluation results. Columns denote median latency; 99th-percentile latency is shown in whiskers.

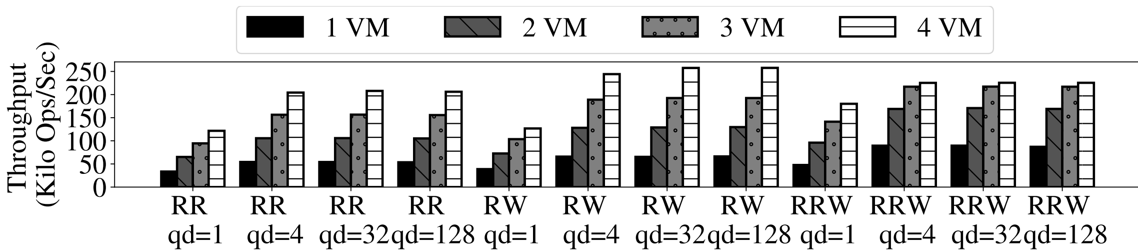

Figure 5.5: NVMetro scalability evaluation results.

Our YCSB benchmark results in Figure 5.6 show little performance variation between all solutions with 1 running job. At 4 parallel jobs, YCSB becomes more I/O-bound and therefore shows more performance variations; while MDev-NVMe and NVMetro stay

[6]Note that our smaller VM size in this experiment prevents direct comparison with the throughput evaluations presented above.

close to native passthrough performance (within approximately 3%), other virtualization tools fall behind, with `vhost-scsi`, SPDK and QEMU being up to 10%, 31% and 49% slower than device passthrough respectively.

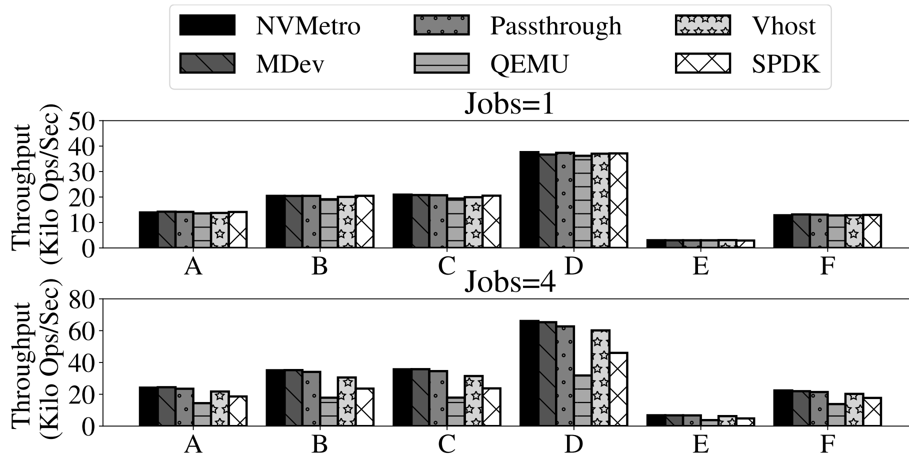

Figure 5.6: YCSB throughput for each workload type (A-F).

5.4.3 Disk encryption evaluations

In this section, we demonstrate the performance of disk encryption using our encryption UIF (with and without SGX) compared to Linux's `dm-crypt` encryption combined with `vhost-scsi` as the virtual storage interface. We also make some comparisons with other, unencrypted scenarios as presented above. Our non-SGX UIF uses 2 worker threads; our SGX UIF uses 1 worker + 1 SGX switchless thread.

Overall, Figure 5.7 shows that our non-SGX UIF outperforms `dm-crypt` at all presented configurations. Notably, at (512B, 16K, 128K)/QD1/1 job, our UIF is up to 1.6×, 1.5× and 1.4× faster than `dm-crypt`. Our solution is even faster with higher parallelism, being 3.2× faster with 16K reads/QD128/4 jobs and 3.7× faster at 128K.

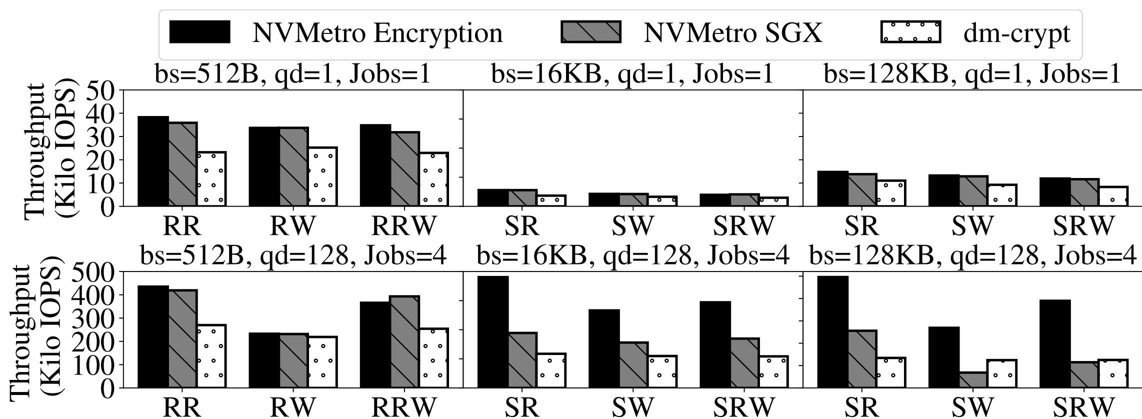

Figure 5.7: Disk encryption evaluations with `fio`.

Our SGX-based encryption UIF performs mostly the same as the non-SGX one, with the exception of 16K/QD128/4 jobs and 128K/QD128/4 jobs being up to 50% and

75% slower than non-SGX on average, as well as 128K writes/QD128/4 jobs being 45% slower than dm-crypt. Both of these results are explained by the lower number of encryption worker threads (as 1 thread is used for switchless enclave calls).

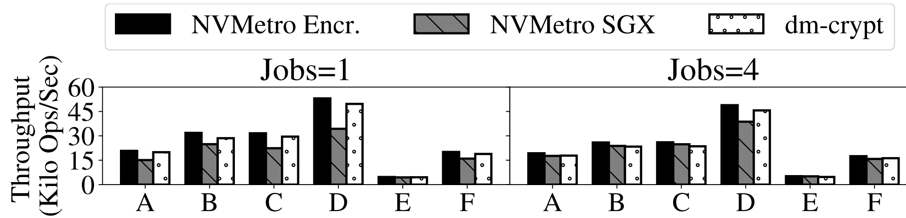

Figure 5.8: Disk encryption evaluations with YCSB.

In Figure 5.8, the YCSB benchmark shows similar performance between our non-SGX UIF and dm-crypt. There is, however, a slightly different performance gap between SGX and non-SGX concerning the number of YCSB jobs. With a single YCSB job, our SGX UIF is up to 35% slower than non-SGX in workload D; however, the SGX implementation gains back some ground at 4 jobs, with the worst-performing workload D only losing 21% performance, while most other workloads become comparable to non-SGX.

5.4.4 Disk replication evaluations

In this section, we compare NVMetro's disk mirroring with Linux's dm-mirror and vhost-scsi on the VM host. In general, both NVMetro replication and dm-mirror perform better at reading than writing; this is easily explained since reads can be directly serviced by the local drive without having to be propagated to the remote. When comparing the two disk mirroring solutions using fio (see Figure 5.9), NVMetro outperforms dm-mirror at all configurations, with a performance advantage of 68%, 220% and 291% at 512B reads/QD1/1 job, 512B reads/QD128/4 jobs and 128K reads/QD128/4 jobs respectively, demonstrating NVMetro's I/O path flexibility in choosing the more efficient data read path.

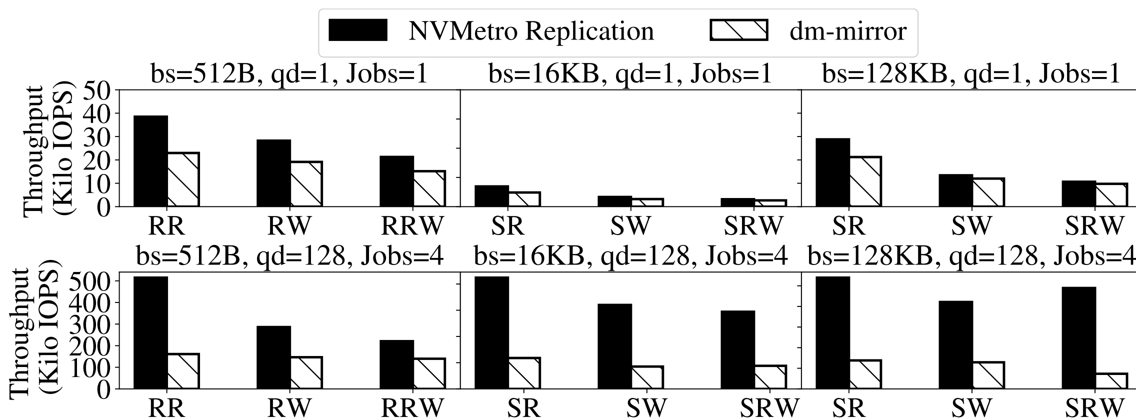

Figure 5.9: Disk replication evaluations with fio.

Figure 5.10 shows our disk replication performance in the YCSB benchmark. In general, our replication function is faster than dm-mirror no matter the workload or

number of parallel jobs. Again our scalability advantages are shown: our replicator performs 2% better at workload D with 1 YCSB job but 17% better with 4 jobs.

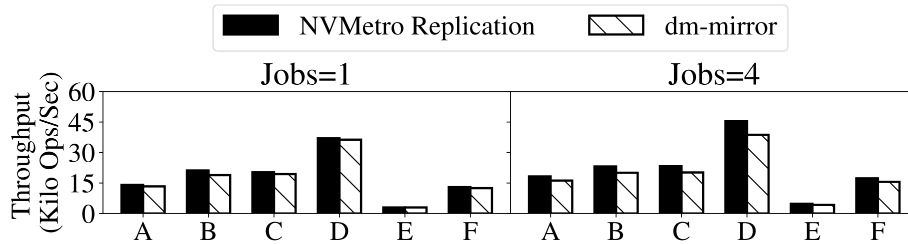

Figure 5.10: Disk replication evaluations with YCSB.

5.4.5 Overhead evaluations

In this section, we compare the CPU consumption of each virtualization method while running the `fio` benchmark under each of the conditions presented above. The CPU consumption is presented in terms of total CPU time used of the whole system, including the VM and any host-side agents.

Basic evaluations (Figure 5.11). Concerning CPU consumption, device passthrough predictably performs the best among all tested configurations. MDev-NVMe, NVMetro and QEMU perform similarly, consuming approximately 85% more total CPU than device passthrough at 512B/QD1/1 job, and ≈ 26% more in the intensive benchmark of 512B/QD128/4 jobs, with the exception of 128KB/QD1/1 job where QEMU uses less CPU than that of the other two. `vhost-scsi` uses less CPU than others still, being the second-lowest CPU-consuming virtualization method, only consuming more CPU than device passthrough. Conversely, SPDK uses more CPU than all other tools, with a ≈ 56% CPU overhead at 512B/QD128/4 jobs. The higher CPU consumption of MDev-NVMe, NVMetro and especially SPDK is explained by these solutions receiving new I/O requests using active polling.

Disk encryption (Figure 5.12). At (512B, 16K, 128K) QD1/1 job, our encryption UIF uses around 2.7×, 2.4× and 2.1× the CPU of `dm-crypt`. While our UIF's CPU utilization is higher than that of `dm-crypt` at lower parallelism, we again gain ground in performance and CPU consumption at higher parallelism: at 4 parallel jobs, NVMetro has approximately the same CPU load as `dm-crypt` in the read benchmark, or even slightly lower at 16K and 128K block sizes.

Our SGX-based UIF has a rather uniform CPU cost at lower parallelism modes: with (512B, 16K, 128K)/QD1/1 job, we use between 10% and 12% more CPU for essentially the same performance. At QD128/4 job, our UIF uses the same amount of CPU due to our maximum CPU constraint.

Disk replication (Figure 5.13). At 512B/QD1/1 job, 512B/QD128/4 jobs and 128K/QD128/4 jobs, NVMetro incurs a CPU cost up to 178%, 36% and 76% higher than

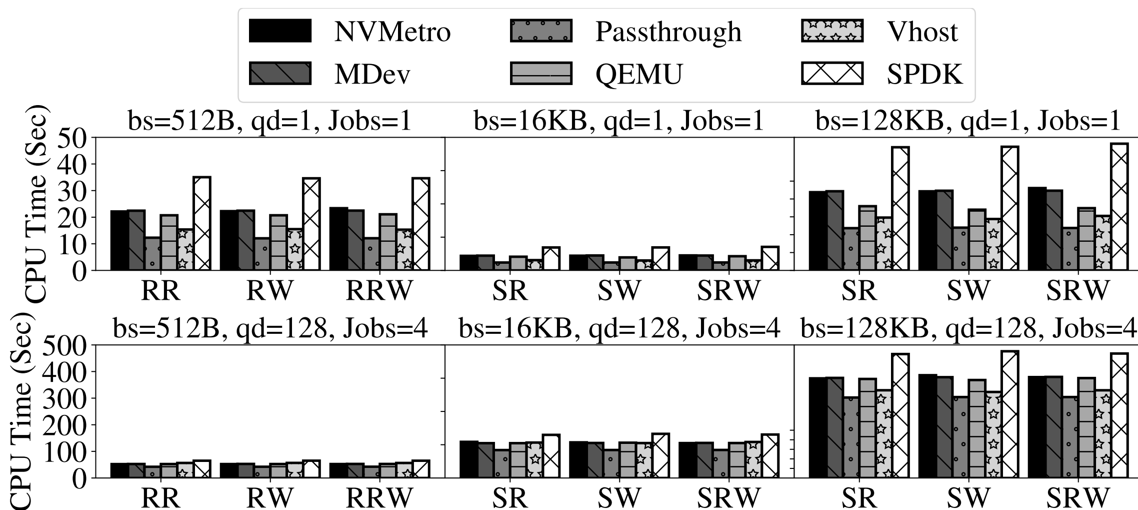

Figure 5.11: CPU consumption of `fio` with basic evaluation.

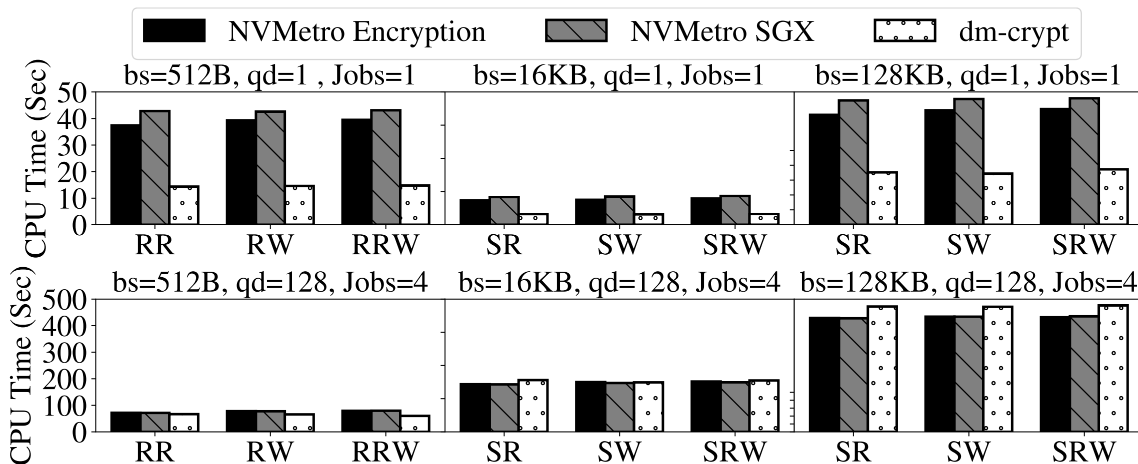

Figure 5.12: CPU consumption of `fio` with disk encryption.

that of `dm-mirror`; nevertheless, this CPU cost is coupled with a higher performance, especially at 128K reads/QD128/4 jobs where we pay 35% more CPU for 291% more throughput, a combination of NVMetro's poll-based I/O and routing of requests to the more efficient I/O path.

5.4.6 NVMetro's flexibility and ease of use in perspective

As we claimed in Section 5.2.2, NVMetro offers a storage function framework that is more flexible and easy to use when compared to existing systems. In this section, we support our claims by analyzing our implementations of the storage functions presented above in contrast with other storage solutions.

Compared to Linux's `vhost-scsi` and device mapper. Linux's in-kernel storage virtualization framework includes two independent components: the `vhost-scsi` facility that provides a `virtio-scsi` storage interface to virtual machines, and the device mapper ("dm" for short) that provides a stackable logic layer on top of storage

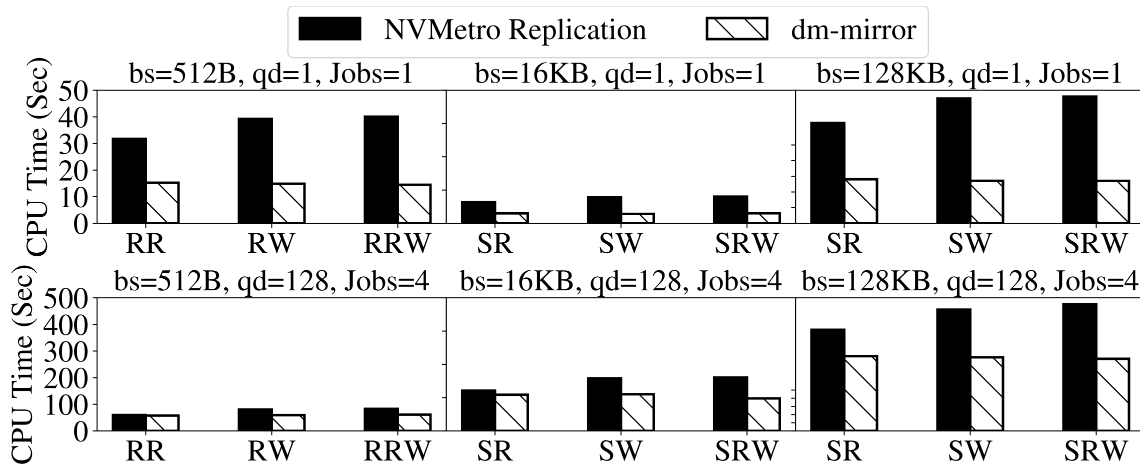

Figure 5.13: CPU consumption of `fio` with disk replication.

devices (similar to FreeBSD's GEOM [113] and Windows's filter drivers). Together, these two give the host control over each VM's access to storage.

Linux's device mapper implements all of its mapping targets as functions inside the kernel, rather than independent programs. These targets can further be stacked as a way to combine simple block mapping functions; however, the use of specific technologies such as Intel SGX poses an additional challenge, as Linux only supports user-mode SGX applications at the moment. In contrast, Intel SGX is easily integrated into our encryption UIF using the corresponding SDK.

Certain patchsets enable communication between userspace and the storage logic layer [114]; in contrast, NVMetro is designed from the ground up to ensure performant communication between the kernel and UIFs through the use of multiple asynchronous queues and adaptive polling. NVMetro's userspace-kernel decoupling enables the implementation of storage functions that serve multiple VMs while reducing the use of costly I/O polling threads. Finally, our request router lets requests bypass UIFs whenever possible thanks to eBPF-coded fast paths. This is apparent by our implementation of the disk replication storage function: only write requests need to be considered by the UIF, while read requests are filtered out by our classifier and directly passed to the underlying disk.

Compared to MDev-NVMe. To reiterate, MDev-NVMe serves as a basis for our implementation of NVMetro. As such, our goal is not to beat MDev-NVMe in raw performance; instead, in addition to MDev-NVMe's fast VM storage queues, NVMetro brings with it an iterative classification and routing component, as well as a pathway for UIFs to communicate with the VMs being serviced. As shown in our evaluations, our additional components did not introduce a significant overhead compared to the existing MDev-NVMe mechanism. A possible alternative is to implement all of the storage logic directly inside the MDev-NVMe module, or to offload it to the device mapper layer; however, this approach is subject to the same limitations as other in-kernel solutions presented above.

Compared to in-VMM virtualization (QEMU) Userspace VMMs such as QEMU are responsible for managing a VM's execution, and for emulating any device needed by the VM. As such, it has full control over a VM's I/O request flow. However, in-VMM virtualization has two significant limitations. Firstly, I/O accesses need to be trapped into the hypervisor kernel before being relayed back to the VMM to be serviced. Moreover, more hypervisor operations are needed to signal the VM of the I/O request status (e.g. by setting virtual interrupts), and to return control from the VMM to the VM's vCPU. Secondly, even with solutions that avoid the above flaw (e.g. Virtio at high QDs), each VMM is responsible for handling its own VM's storage requests. In scenarios with high VM densities, this leads to large amounts of CPU time and context switches being wasted for the sake of handling I/O separately on each VM, which limits the scalability of this solution.

Compared to SPDK SPDK is comparable to NVMetro as a set of tools and libraries for writing user-mode storage applications. It possesses many similar capabilities: stackable storage logic, ability to colocate multiple storage targets in one process, and so on. However, NVMetro provides two main benefits compared to SPDK. Firstly, unlike SPDK, NVMetro does not require exclusive assignment of a storage device; host applications and other VMs can easily share the same device while using the familiar POSIX API. Secondly, NVMetro can be gradually applied to I/O requests as requirements evolve. Particularly, in our disk replication example, the storage function developer does not need to be concerned about hardware internals, or the handling of irrelevant requests and commands; relevant requests are selected in eBPF, and our UIFs use standard POSIX APIs to communicate with the NVMetro router. Combined with our simplified UIF framework, NVMetro significantly reduces the complexity of storage functions.

5.5 Related works

General computational storage architecture. SNIA's Computational Storage Architecture and Programming Model [115] defines a general structure of computational storage applications, where different kinds of storage engines, including eBPF-based ones, can be embedded into various device classes. The authors also define several types of computational storage functions that can be used on these engines.

Virtual storage providers. SPDK [105] is a fast storage framework based on top of the NVMe protocol. In the same vein as DPDK, it uses an userspace driver via device passthrough to deliver various services, including the Vhost service for providing virtual disks to VMs. SPDK Vhost-NVMe [116] builds a virtual NVMe interface on top of SPDK with an optimized I/O path. Direct-Virtio [117] proposes an userspace QEMU-based Virtio target with predictive polling.

NVMe Direct [118] introduces a NVMe-specific, queue-based API similar to that of the RDMA Connection Manager for RDMA network devices to grant userspace programs direct access to NVMe's high-performance queues. NVMe Direct 2.0 [119] is

a preloadable shared library that shadows I/O functions (similar to network offloading libraries such as `libsdp` and Mellanox's `libvma`) to improve performance without needing to change the application code.

Vhost is Linux's paravirtualized device protocol based on the Virtio specification to provide fast and efficient networking and storage services for KVM guests. Vhost offloads I/O processing to either the host kernel itself [120] or to an external process via `vhost-user` [121]. MDev-NVMe [110] describes a NVMe virtualization layer based on active polling to improve I/O throughput and reduce latency. Notably, MDev-NVMe bypasses many subsystems of the Linux kernel to reduce the cost of each I/O operation. H-NVMe [122] optimizes the hypervisor's storage stack by offering two new modes, namely "Parallel Queue Mode" that improves parallelism, and "Direct Access Mode" that grants trusted VMs direct access to the device queue. FAST I/O [123] proposes quality-of-service controls of I/O operations on NVMe devices by submitting high-priority requests directly to an admin NVMe queue, therefore bypassing the operating system-level queues, and by writing request data to the drive's host memory buffer.

NVMetro also belongs to the category of virtual storage providers. The advantage of NVMetro compared to others is a combination of kernel- and userspace-based logic to allow developers to quickly and easily customize their virtual I/O path per-request depending on their use case.

Sandboxed-bytecode (eBPF, WebAssembly)-based solutions. Most works on this topic propose the use of eBPF to offload computing tasks to local storage agents. Zhong et al. [13] investigate the feasibility of inserting BPF hooks into Linux's storage stack in order to provide extra functionalities, e.g. tree lookups. Griffin [124] envisions a set of APIs using eBPF to add logic to storage applications running on edge computing nodes. Kourtis et al. [125] follow in the same line by running eBPF on top of NBD, and propose ways to use eBPF for key-value stores and SQL offloading. Huang and Paradies [126] compare eBPF and WebAssembly's various aspects (safety, compatibility, performance, etc.), and recommend how to develop these technologies for storage offloading purposes.

Generally speaking, these solutions suggest extending eBPF or replacing it with another runtime (e.g. WebAssembly), citing eBPF's current limitations. In contrast, NVMetro requires no change to the kernel's eBPF implementation, as the eBPF code only serves as a first-line classifier inside the request router; complex operations can be offloaded to UIFs.

Hardware-based solutions. In this category, LeapIO [127] presents a new storage stack based on offloading virtualization tasks onto on-disk processors coupled with smart memory and NIC sharing to improve performance. FastPath [128] adds a FPGA-based computing engine between the host and storage device, then exposes an API to the FPGA to offer a fast path to applications needing high I/O performance. FastPath_MP [129] extends FastPath with support for multiple I/O queues to take advantage of the parallelism offered by NVMe devices. FVM [130] is a similar solution that

interposes hardware NVMe devices with FPGA to aid virtualization by performing I/O queue emulation and storage address translation.

5.6 Summary

In this chapter, we introduced NVMetro, a flexible I/O virtualization framework that eases the development of sophisticated storage functions. NVMetro builds upon a mediated NVMe interface with a combination of fast eBPF-based I/O classifier/router and userspace I/O functions; by allowing the creation of multiple I/O paths of varying characteristics, NVMetro ensures that storage function remains fast, secure and manageable regardless of the use case. We described the design criteria that lead to NVMetro's design, and elaborated on the design and development of several sample storage use cases. We evaluated NVMetro in comparison to existing systems, and showed the performance, scalability and simplicity of our storage function implementations using multiple benchmarks, thus demonstrating the flexibility of our framework.

Chapter 6

Conclusion and perspectives

The previous chapters have investigated the various degrees of freedom and associated tradeoffs of virtualized infrastructure configuration. To summarize, Walk-In explored three expressions of the idea of *flexibility*:

- Chapter 3 established the value of operating multiple hypervisors on the same VM host cluster in avoiding active vulnerabilities, using the concept of hypervisor transplant implemented through an unified VM state representation. At the same time, HyperTP offers two complementary transplant methods that balance transplant time, downtime and memory overhead, and can be freely chosen for enhancing the security of hypervisor deployments.
- Chapter 4 focused on the loss of transparency over system topology when running I/O-heavy applications on virtual machines. We introduced a comprehensive solution to this issue in both the hypervisor and guest environment, which works by dynamically sharing a physical resource (i.e. I/O home node resource).
- Chapter 5 presented NVMetro, a virtual I/O framework built from the ground up for flexibility and performance based on the NVMe specification. NVMetro's classifier/router and UIF framework offer multiple I/O paths to the storage function developer, and intelligently switches between them during an I/O request's lifecycle while making explicit the tradeoffs between these paths.

With Walk-In, we have shown that **the various challenges of virtualized systems can be solved through measured application of flexibility in their configuration spaces**. Keeping this idea in mind, we suggest several future research directions:

HyperTP for more than just momentary vulnerability protection. Our goal is to further shrink the patch gap of hypervisors beyond the point of vulnerability reporting. We propose the active and constant application of HyperTP on running VMs, where VM states are continuously circulated within the VM cluster. With this system, any attack on a particular hypervisor will be negated by automatically switching to another already-running hypervisor, without needing explicit knowledge of a vulnerability.

NVMetro as a platform for better virtual storage. NVMetro's flexible and performant design serves as a technical basis for exploring the challenges of VM storage, such as enabling efficient migration, data replication, and controlling access to sensitive data. We foresee two directions of development in this pathway: firstly, improving NVMetro's core mechanisms to serve new and efficient storage functions; and secondly, implementing these functions to better serve VMs.

Flexibility beyond the VM infrastructure. In current years, we have observed the appearance of multiple innovative forms of compute virtualization, such as hyper-converged architectures, overlay networks, serverless computing, and so on. As the solutions get more sophisticated, so do their configuration space; therefore, in the long term, our goal is to investigate these computing paradigms to better understand their specificities and tradeoffs, and to devise new ways to exploit their flexibility.

Common acronyms

Notation	Description
eBPF	(also *Extended BPF*) Extended Berkeley Packet Filter *compare with* Berkeley Packet Filter (BPF)
FaaS	function-as-a-service
HVM	hardware virtual machine (Xen)
IaaS	infrastructure-as-a-service
IOMMU	I/O memory management unit *compare with* memory management unit (MMU)
IPI	interprocessor interrupt
KLOC	1000 lines of code
NUIOA	non-uniform I/O access
NUMA	non-uniform memory access
NVMe	(also *NVM Express*) Non-Volatile Memory Express
PCIe	(also *PCI Express*) Peripheral Component Interconnect Express
PV	paravirtualization; paravirtualized
RDMA	remote direct memory access
UIF	userspace I/O function
UISR	Unified Intermediate State Representation
vCPU	virtual CPU
VFIO	Virtual Function I/O (Linux)
VM	virtual machine
VMM	virtual machine monitor

Bibliography

[1] Tu Dinh Ngoc, Boris Teabe, Alain Tchana, Gilles Muller, and Daniel Hagimont. Mitigating vulnerability windows with hypervisor transplant. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 162–177, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383349. doi:10.1145/3447786.3456235.

[2] Tu Dinh Ngoc, Boris Teabe, Daniel Hagimont, and Georges Da Costa. Optimized resource allocation on virtualized non-uniform I/O architectures. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 432–441. IEEE, 2022. doi:10.1109/CCGrid54584.2022.00053.

[3] Tu Dinh Ngoc, Boris Teabe, Alain Tchana, Gilles Muller, and Daniel Hagimont. HyperTP: A unified approach for live hypervisor replacement in datacenters. *Journal of Parallel and Distributed Computing*, page 104733, 2023. doi:10.1016/j.jpdc.2023.104733.

[4] Tu Dinh Ngoc, Boris Teabe, Georges Da Costa, and Daniel Hagimont. Flexible NVMe request routing for virtual machines. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024.

[5] Brice Ekane, Tu Dinh Ngoc, Boris Teabe, Daniel Hagimont, and Noel De Palma. FlexVF: Adaptive network device services in a virtualized environment. *Future Generation Computer Systems*, 127:14–22, 2022. doi:10.1016/j.future.2021.08.011.

[6] Jean-Baptiste Decourcelle, Tu Dinh Ngoc, Boris Teabe, and Daniel Hagimont. Fast VM replication on heterogeneous hypervisors for robust fault tolerance. In *Proceedings of the 24th International Middleware Conference*, Middleware '23, pages 15–28, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701771. doi:10.1145/3590140.3592849.

[7] National Cyber Security Centre, UK. The problems with patching, 2019. URL `https://www.ncsc.gov.uk/blog-post/the-problems-with-patching`.

[8] Nesara Dissanayake, Asangi Jayatilaka, Mansooreh Zahedi, and M Ali Babar. Software security patch management - a systematic literature review of challenges, approaches, tools and practices. *Information and Software Technology*, 144:106771, 2022.

[9] Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. Fast and scalable VMM live upgrade in large cloud infrastructure. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–105, 2019.

[10] Gauthier Voron, Gaël Thomas, Vivien Quéma, and Pierre Sens. An interface to implement NUMA policies in the Xen hypervisor. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 453–467, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349383. doi:10.1145/3064176.3064196.

[11] Bao Bui, Djob Mvondo, Boris Teabe, Kevin Jiokeng, Lavoisier Wapet, Alain Tchana, Gaël Thomas, Daniel Hagimont, Gilles Muller, and Noel DePalma. When eXtended Para-Virtualization (XPV) meets NUMA. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. doi:10.1145/3302424.3303960.

[12] Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. Reducing world switches in virtualized environment with flexible cross-world calls. *ACM SIGARCH Computer Architecture News*, 43(3S):375–387, 2015.

[13] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.

[14] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[15] Andrew Whitaker, Marianne Shaw, Steven D Gribble, et al. Denali: Lightweight virtual machines for distributed and networked applications. 2002.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[17] Steve Rutherford. Performant security hardening, 2016. URL `https://www.linux-kvm.org/images/3/3d/01x02-Steve_Rutherford-Performant_Security_Hardening_of_KVM.pdf`.

[18] Robert P Goldberg. *Architectural principles for virtual computer systems*. PhD thesis, Harvard University Cambridge, MA, 1973.

[19] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y Wang. Bringing virtualization to the x86 architecture with the original VMWare Workstation. *ACM Transactions on Computer Systems (TOCS)*, 30(4): 1–51, 2012.

[20] VMware, Inc. Understanding full virtualization, paravirtualization, and hardware assist, 2007. URL `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf`.

[21] QEMU documentation authors. QEMU documentation - TCG, 2019. URL `https://wiki.qemu.org/Documentation/TCG`.

[22] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[23] Intel Corp. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 3C, page 24-2. 2023.

[24] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha 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*, pages 26–35, 2008.

[25] Jayneel Gandhi, Mark D Hill, and Michael M Swift. Agile paging: Exceeding the best of nested and shadow paging. *ACM SIGARCH Computer Architecture News*, 44(3):707–718, 2016.

[26] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. Everything you should know about Intel SGX performance on virtualized systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(1):1–21, 2019. doi:10.1145/3322205.3311076.

[27] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[28] Brendan Gregg. Xen modes: What color is your Xen?, 2014. URL `https://www.brendangregg.com/blog/2014-05-07/what-color-is-your-xen.html`.

[29] Brice Goglin and Stephanie Moreaud. Dodging non-uniform I/O access in hierarchical collective operations for multicore clusters. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, page 788–794, USA, 2011. IEEE Computer Society. ISBN 9780769545776. doi:10.1109/IPDPS.2011.222.

[30] NVM Express, Inc. NVM Express specifications, 2021. URL `https://nvmexpress.org/specifications/`.

[31] Intel Corp. Intel Optane SSD P5800X Series, 2021. URL `https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html`.

[32] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, 1993.

[33] The kernel development community. BPF documentation - the Linux kernel documentation, 2021. URL https://www.kernel.org/doc/html/latest/bpf/index.html.

[34] Cilium. BPF and XDP reference guide, 2021. URL https://docs.cilium.io/en/latest/bpf/.

[35] kpatch authors. kpatch patch author guide, 2024. URL https://github.com/dynup/kpatch/blob/master/doc/patch-author-guide.md.

[36] Google LLC. LLpatch: LLVM-based kernel livepatch generation, 2024. URL https://github.com/google/LLpatch?tab=readme-ov-file#notes.

[37] Microsoft Corp. Maintenance for virtual machines in Azure, 2022. URL https://docs.microsoft.com/en-us/azure/virtual-machines/maintenance-and-updates.

[38] Andrea Segalini, Dino Lopez Pacheco, Guillaume Urvoy-Keller, Fabien Hermenier, and Quentin Jacquemart. Hy-FiX: Fast in-place upgrades of KVM hypervisors. *IEEE Transactions on Cloud Computing*, 10(4):2679–2690, 2022. doi:10.1109/TCC.2021.3056590.

[39] Sun Microsystems, Inc. XDR: External data representation standard, 1987. URL https://tools.ietf.org/html/rfc1014.

[40] Hardik Bagdi, Rohith Kugve, and Kartik Gopalan. Hyperfresh: Live refresh of hypervisors using nested virtualization. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–8, 2017.

[41] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35. Springer, 2005.

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. Meltdown: Reading kernel memory from user space. *Communications of the ACM*, 63(6):46–56, 2020.

[43] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[44] NIST. National Vulnerability Database - vulnerability metrics. URL https://nvd.nist.gov/vuln-metrics/cvss.

[45] Vladimir Davydov. pram: persistent over-kexec memory file system, 2013. URL https://lists.openvz.org/pipermail/criu/2013-July/009877.html.

[46] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi:10.1145/3132747.3132772.

[47] Fabien Hermenier, Julia Lawall, and Gilles Muller. Btrplace: A flexible consolidation manager for highly available applications. *IEEE Transactions on dependable and Secure Computing*, 10(5):273–286, 2013.

[48] CVE-2018-18883. CVE-2018-18883, 2018. URL `https://www.cvedetails.com/cve/CVE-2018-18883/`.

[49] Jake Edge. OpenBSD kernel address randomized link, 2017. URL `https://lwn.net/Articles/727697/`.

[50] Alexander Lobakin. Function granular KASLR, 2021. URL `https://lore.kernel.org/lkml/20211223002209.1092165-1-alexandr.lobakin@intel.com/`.

[51] Trusted Computing Group. TCG D-RTM architecture, 2013. URL `https://trustedcomputinggroup.org/resource/d-rtm-architecture-specification/`.

[52] Intel Corp. Intel Trusted Execution Technology (Intel TXT) overview, 2022. URL `https://www.intel.com/content/www/us/en/support/articles/000025873/processors.html`.

[53] Kenichi Kourai and Hiroki Ooba. Zero-copy migration for lightweight software rejuvenation of virtualized systems. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, pages 1–8, 2015.

[54] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption, 2016. URL `https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v9-Public.pdf`.

[55] Jiří Kosina, Petr Mládek, Vojtěch Pavlík, and Jiri Slaby. kGraft: Live patching of the Linux kernel, 2014. URL `https://kernel-recipes.org/en/2014/kgraft-live-patching-of-the-linux-kernel/`.

[56] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 187–198, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584829. doi:10.1145/1519065.1519085.

[57] Hai Jin, Li Deng, Song Wu, Xuanhua Shi, and Xiaodong Pan. Live virtual machine migration with adaptive, memory compression. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[58] Umesh Deshpande, Unmesh Kulkarni, and Kartik Gopalan. Inter-rack live migration of multiple virtual machines. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, pages 19–26, 2012.

[59] Konstantinos Tsakalozos, Vasilis Verroios, Mema Roussopoulos, and Alex Delis. Live VM migration under time-constraints in share-nothing IaaS-clouds. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2285–2298, 2017.

[60] Pengcheng Liu, Ziye Yang, Xiang Song, Yixun Zhou, Haibo Chen, and Binyu Zang. Heterogeneous live migration of virtual machines. In *International Workshop on Virtualization Technology (IWVT'08)*, 2008.

[61] Hiroshi Yamada and Kenji Kono. Traveling forward in time to newer operating systems using ShadowReboot. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, 2013.

[62] Ibrahim Ejdayid A Mansour, Kendra Cooper, and Hamid Bouchachia. Effective live cloud migration. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 334–339. IEEE, 2016.

[63] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving Xen security through disaggregation. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595937964. doi:10.1145/1346256.1346278.

[64] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 401–412, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309486. doi:10.1145/2046707.2046754.

[65] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[66] Le Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. URL `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/deconstructing-xen/`.

[67] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In

Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 203–216. ACM, 2011. doi:10.1145/2043556.2043576.

[68] Weidong Shi, JongHyuk Lee, Taeweon Suh, Dong Hyuk Woo, and Xinwen Zhang. Architectural support of multiple hypervisors over single platform for enhancing cloud computing security. In *Proceedings of the 9th Conference on Computing Frontiers*, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312158. doi:10.1145/2212908.2212920.

[69] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives, 2012.

[70] Idris Winarno, Takeshi Okamoto, Yoshikazu Hata, and Yoshiteru Ishida. Increasing the diversity of resilient server using multiple virtualization engines. *Procedia Computer Science*, 96:1701–1709, 2016.

[71] Yuesheng Tan, Dengliang Luo, and Jingyu Wang. CC-VIT: Virtualization intrusion tolerance based on cloud computing. In *2010 2nd International Conference on Information Engineering and Computer Science*, pages 1–6. IEEE, 2010.

[72] K. Kourai and S. Chiba. Fast software rejuvenation of virtual machine monitors. *IEEE Transactions on Dependable and Secure Computing*, 8(6):839–851, 2011.

[73] Alex Depoutovitch and Michael Stumm. Otherworld: giving applications a chance to survive OS kernel crashes. In *Proceedings of the 5th European conference on Computer systems*, pages 181–194, 2010.

[74] Mark Russinovich, Naga Govindaraju, Melur Raghuraman, David Hepkin, Jamie Schwartz, and Arun Kishan. Virtual machine preserving host updates for zero day patching in public cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 114–129, 2021.

[75] M. Le and Y. Tamir. Applying microreboot to system software. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 11–20, 2012.

[76] X. Xu and H. H. Huang. DualVisor: Redundant hypervisor execution for achieving hardware error resilience in datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 485–494, 2015.

[77] D. Zhou and Y. Tamir. Fast hypervisor recovery without reboot. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 115–126, 2018.

[78] F. Cerveira, R. Barbosa, and H. Madeira. Fast Local VM Migration Against Hypervisor Corruption. In *2019 15th European Dependable Computing Conference (EDCC)*, pages 97–102, 2019.

[79] S. Van Doren. Abstract - hoti 2019: Compute express link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 18–18, Los Alamitos, CA, USA, aug 2019. IEEE Computer Society. doi:10.1109/HOTI.2019.00017. URL https://doi.ieeecomputersociety.org/10.1109/HOTI.2019.00017.

[80] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern numa systems. *Commun. ACM*, 58(12):59–66, November 2015. ISSN 0001-0782. doi:10.1145/2814328.

[81] Libnuma. A NUMA API for LINUX*., 2013. URL http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf.

[82] Tiago Rosado and Jorge Bernardino. An overview of OpenStack architecture. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, IDEAS '14, page 366–367, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326278. doi:10.1145/2628194.2628195.

[83] OpenStack. I/O (PCIe) based NUMA scheduling, 2021. URL https://specs.openstack.org/openstack/nova-specs/specs/kilo/implemented/input-output-based-numa-scheduling.html.

[84] Mellanox Technologies. Mellanox/sockperf: Network bench-marking utility, 2021. URL https://github.com/Mellanox/sockperf.

[85] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003. ISSN 0163-5980. doi:10.1145/1165389.945462.

[86] Libvma. Libvma., 2021. URL https://github.com/Mellanox/libvma.

[87] Alexey Kopytov. Sysbench: a system performance benchmark. URL https://github.com/akopytov/sysbench.

[88] Perftest. Perftest Package., 2013. URL https://community.mellanox.com/s/article/perftest-package.

[89] Ahmed Soliman. *Getting Started with Memcached*. Packt Publishing, 2013. ISBN 1782163220.

[90] Jens Axboe. Flexible I/O tester, 2021. URL https://github.com/axboe/fio.

[91] Oracle. Soft affinity - when hard partitioning is too much, 2019. URL https://blogs.oracle.com/linux/post/soft-affinity-when-hard-partitioning-is-too-much.

[92] Daniel Berrangé. Openstack performance optimization: Numa, largepages & cpu pinning, 2014. URL https://www.linux-kvm.org/images/0/0b/03x03-Openstackpdf.pdf.

[93] Y. Ren, T. Li, D. Yu, S. Jin, and T. Robertazzi. Design, implementation, and evaluation of a NUMA-aware cache for iSCSI storage servers. *IEEE Transactions on Parallel and Distributed Systems*, 26(2):413–422, 2015. doi:10.1109/TPDS.2014.2311817.

[94] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. High-speed query processing over high-speed networks. *Proc. VLDB Endow.*, 9(4): 228–239, December 2015. ISSN 2150-8097. doi:10.14778/2856318.2856319.

[95] VMware Technical Publications. Tuning vCloud NFV for data plane intensive workloads, OpenStack edition., 2019. URL `https://docs.vmware.com/en/VMware-vCloud-NFV-OpenStack-Edition/3.0/vmwa-vcloud-nfv30-performance-tunning.pdf`.

[96] Amitabha Banerjee, Rishi Mehta, and Zach Shen. NUMA aware I/O in virtualized systems. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, HOTI '15, page 10–17, USA, 2015. IEEE Computer Society. ISBN 9781467391603. doi:10.1109/HOTI.2015.17.

[97] Lance Shelton. High performance I/O with NUMA systems in Linux., 2013. URL `https://events.static.linuxfound.org/sites/events/files/eeus13_shelton.pdf`.

[98] D. Mvondo, B. Teabe, A. Tchana, D. Hagimont, and N. De Palma. Closer: A new design principle for the privileged virtual machine os. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 49–60, 2019. doi:10.1109/MASCOTS.2019.00016.

[99] B. Teabe, A. Tchana, and D. Hagimont. Billing system cpu time on individual vm. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 493–496, 2016. doi:10.1109/CCGrid.2016.76.

[100] Red Hat, Inc. libvirt NUMA tuning. URL `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-numa-numa_and_libvirt`.

[101] Jeff Squyres. Process and memory affinity: why do you care? High Performance Computing Networking., 2013. URL `http://blogs.cisco.com/performance/process-and-memory-affinity-why-do-you-care`.

[102] Stéphanie Moreaud, Brice Goglin, and Raymond Namyst. Adaptive MPI multirail tuning for non-uniform input/output access. In *European MPI Users' Group Meeting*, pages 239–248. Springer, 2010.

[103] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. Ioctopus: Outsmarting

nonuniform dma. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 101–115, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi:10.1145/3373376.3378509.

[104] NVIDIA. NVIDIA Mellanox Socket Direct adapters, 2021. URL `https://www.nvidia.com/en-us/networking/ethernet/socket-direct/`.

[105] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[106] DPDK Project. Kernel NIC interface, 2021. URL `https://doc.dpdk.org/guides/prog_guide/kernel_nic_interface.html`.

[107] Miklos Szeredi. FUSE: Filesystem in userspace, 2010. URL `https://github.com/libfuse/libfuse`.

[108] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, 2017.

[109] Cloudflare. Why we use the Linux kernel's TCP stack, 2016. URL `https://blog.cloudflare.com/why-we-use-the-linux-kernels-tcp-stack/`.

[110] Bo Peng, Jianguo Yao, Yaozu Dong, and Haibing Guan. MDev-NVMe: Mediated pass-through NVMe virtualization solution with adaptive polling. *IEEE Transactions on Computers*, 71(2):251–265, 2020.

[111] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[112] Maxim Levitsky. NVME VFIO mediated device, 2019. URL `https://lkml.org/lkml/2019/3/19/458`.

[113] Poul-Henning Kamp. GEOM tutorial, 2004. URL `https://papers.freebsd.org/2004/phk-geom-tutorial.files/bsdcan-04.slides.geomtut.pdf`.

[114] Palmer Dabbelt. dm-user: New target that proxies BIOs to userspace, 2020. URL `https://lwn.net/Articles/838986/`.

[115] SNIA. Computational storage architecture and programming model, 2021. URL `https://www.snia.org/sites/default/files/technical_work/PublicReview/SNIA-Computational-Storage-Architecture-and-Programming-Model-0.8R0-2021.06.09.pdf`.

[116] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. SPDK Vhost-NVMe: Accelerating I/Os in virtual machines on NVMe SSDs via user space Vhost target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 67–76. IEEE, 2018.

[117] Sewoog Kim, Heekwon Park, and Jongmoo Choi. Direct-Virtio: A new direct virtualized I/O framework for NVMe SSDs. *Electronics*, 10(17):2058, 2021.

[118] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[119] Jin-Soo Kim Hyeong-Jun Kim and Jin-Soo Kim. NVMeDirect 2.0: An enhanced user-space I/O framework for NVMe SSDs. *Flash Memory Summit*, 2017.

[120] Red Hat, Inc. Deep dive into Virtio-networking and vhost-net, 2019. URL `https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net`.

[121] Red Hat, Inc. A journey to the vhost-users realm, 2019. URL `https://www.redhat.com/en/blog/journey-vhost-users-realm`.

[122] Zhengyu Yang, Morteza Hoseinzadeh, Ping Wong, John Artoux, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. H-NVMe: a hybrid framework of NVMe-based storage system in cloud computing environment. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2017.

[123] Kyusik Kim, Seongmin Kim, and Taeseok Kim. FAST I/O: QoS supports for urgent I/Os in NVMe SSDs. In *Proceedings of the 2020 5th International Conference on Intelligent Information Technology*, pages 146–151, 2020.

[124] Giulia Frascaria, Animesh Trivedi, and Lin Wang. A case for a programmable edge storage middleware. *arXiv preprint arXiv:2111.14720*, 2021.

[125] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. Safe and efficient remote application code execution on disaggregated NVM storage with eBPF. *arXiv preprint arXiv:2002.11528*, 2020.

[126] Wenjun Huang and Marcus Paradies. An evaluation of WebAssembly and eBPF as offloading mechanisms in the context of computational storage. *arXiv preprint arXiv:2111.01947*, 2021.

[127] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. LeapIO: Efficient and portable virtual NVMe storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 591–605, 2020.

[128] Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, and Mikel Luján. FastPath: towards wire-speed NVMe SSDs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 170–1707. IEEE, 2018.

[129] Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, and Mikel Luján. FastPath_MP: Low overhead & energy-efficient FPGA-based storage multi-paths. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–23, 2020.

[130] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. FVM: FPGA-assisted virtual device emulation for fast, scalable, and flexible storage virtualization. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 955–971, 2020.

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Université de Toulouse

Titre : Walk-In : interfaces de virtualisation flexibles pour la performance et la sécurité dans les datacenters modernes

Mots clés : Hyperviseurs, Virtualisation, Sécurité, Performance

Résumé : La virtualisation est un outil puissant conférant de nombreuses avantages pour la sécurité, l'efficacité et la gestion des systèmes informatiques. Par conséquent, la virtualisation est largement utilisée dans les infrastructures informatiques modernes. Cependant, les couches materielles et logicielles supplémentaires posent de nouveaux défis aux administrateurs systèmes. Dans cette thèse, nous présentons notre analyse des défis liés à la virtualisation. Nous constatons que les nombreux composants de virtualisation doivent être tenus à jour, ce qui complique fortement la maintenance logicielle. Deuxièmement, nous remarquons que la virtualisation masque les spécificités du matériel. Enfin, elle réduit les performances des systèmes du fait de la coordination entre les logiciels impliqués par la virtualisation. Nous étudions trois approches pour relever les défis présentés ci-dessus en augmentant la flexibilité de la pile de logiciels de virtualisation. - Notre première contribution concerne la maintenabilité et la sécurité des plateformes de machines virtuelles en tenant compte du besoin de les maintenir à jour. Nous présentons notre cadre « HyperTP » pour la mise à jour des hyperviseurs et l'attenuation des vulnérabilités pendant le délai entre leur découverte et leur correction. - Notre deuxième contribution concerne la perte de performance résultant du manque de visibilité des topologies NUMA dans les machines virtuelles. Nous étudions la performance des charges de travail d'E/S virtuelles afin de déterminer une politique d'allocation des ressources pour l'optimisation des E/S. - Notre troisième contribution concerne la virtualisation des systèmes de stockage à haute performance. Nous présentons la plateforme de stockage NVM-Router, qui permet la réalisation de fonctions de stockage performantes grâce à sa flexibilité et à sa facilité d'utilisation. En résumé, nos contributions montrent les compromis présents dans les configurations des machines virtuelles, ainsi que la réduction des coûts de virtualisation par la manipulation dynamique de ces configurations.

Title: Walk-In: flexible virtualization APIs for performance and security in the modern datacenter

Key words: Hypervisors, Virtualization, Security, Performance

Abstract: Virtualization is a powerful tool that brings numerous benefits for the security, efficiency and management of computer systems. Modern infrastructure therefore makes heavy use of virtualization in almost every software component. However, the extra hardware and software layers present various challenges to the system operator. In this work, we analyze and identify the challenges relevant to virtualization. Firstly, we observe a complexification of maintenance from the numerous software layers that must be constantly updated. Secondly, we notice a lack of transparency on details of the underlying infrastructure from virtualization. Thirdly, virtualization has a damaging effect on system performance, stemming from how the layers of virtualization have to be navigated during operation. We explore three approaches of solving the challenges of virtualization through adding flexibility into the virtualization stack. - Our first contribution tackles the issue of maintainability and security of virtual machine platforms caused by the need to keep these platforms up-to-date. We introduce HyperTP, a framework based on the hypervisor transplant concept for updating hypervisors and mitigating vulnerabilities. - Our second contribution focuses on performance loss resulting from the lack of visibility of non-uniform memory access (NUMA) topologies on virtual machines. We thoroughly evaluate I/O workloads on virtual machines running on NUMA architectures, and implement a unified hypervisor-VM resource allocation strategy for optimizing virtual I/O on said architectures. - For our third work, we focus our attention on high-performance storage subsystems for virtualization purposes. We present NVM-Router, a flexible yet easy to use virtual storage platform that supports the implementation of fast yet efficient storage functions. Together, our solutions demonstrate the tradeoffs present in the configuration spaces of virtual machine deployments, as well as how to reduce virtualization overhead through dynamic adjustment of these configurations.