Beyond virtual machine migration for resources optimization in highly consolidated data centers

Andrea Segalini

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Alternatives à la migration de machines virtuelles pour l'optimisation des ressources dans les centres informatiques hautement consolidés

Andrea SEGALINI

Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S)

Présentée en vue de l'obtention du grade de docteur en INFORMATIQUE **d'**Université Côte d'Azur

Dirigée par : Guillaume Urvoy-Keller
Co-encadrée par : Dino Lopez-Pacheco

Soutenue le : 29/11/2021

Devant le jury, composé de :
Anne-Cécile Orgerie, Chargée de Recherche, CNRS
Daniel Hagimont, Prof., INPT/ENSEEIHT
Dino Lopez-Pacheco, MCR, Université Côte d'Azur
Pietro Michiardi, Prof., EURECOM (*Président*)
Gaël Thomas, Prof., Telecom SudParis/IP Paris
Guillaume Urvoy-Keller, Prof., Université Côte d'Azur

Université Côte d'Azur

DOCTORAL SCHOOL STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

P H D T H E S I S

Andrea SEGALINI

Beyond Virtual Machine Migration for Resources Optimization in Highly Consolidated Data Centers

Thesis Advisor: Guillaume URVOY-KELLER

Thesis Co-advisor: Dino LOPEZ-PACHECO

Prepared at Laboratoire d'Informatique, Signaux et Systèmes
de Sophia Antipolis (I3S)

Defended on 29/11/2021

Jury :

Reviewers :	Daniel HAGIMONT	Prof. at INPT/ENSEEIHT
	Gaël THOMAS	Prof. at Telecom SudParis/IP Paris
Advisor :	Guillaume URVOY-KELLER	Prof. at Université Cote d'Azur
Co-advisor :	Dino LOPEZ-PACHECO	Assoc. Prof. at Université Côte d'Azur
Examiner :	Anne-Cécile ORGERIE	Research Scientist at CNRS
President :	Pietro MICHIARDI	Prof. at EURECOM

Résumé

La virtualisation est une technologie de première importance dans les centres informatiques (*datacenters*). Elle fournit deux mécanismes clés, les machines virtuelles et la migration, qui permettent de maximiser l'utilisation des ressources pour réduire les dépenses d'investissement. Dans cette thèse, nous avons identifié et étudié deux contextes où la migration traditionnelle ne parvient pas à fournir les outils optimaux pour utiliser au mieux les ressources disponibles dans un cluster : les machines virtuelles inactives et les mises à jour à grande échelle des hyperviseurs.

Les machines virtuelles inactives verrouillent en permanence les ressources qui leur sont attribuées uniquement dans l'attente des (rares) demandes des utilisateurs. Ainsi, alors qu'elles sont la plupart du temps inactifs, elles ne peuvent pas être arrêtées, ce qui libérerait des ressources pour des services plus demandeurs. Pour résoudre ce problème, nous proposons SEaMLESS, une solution qui exploite une nouvelle forme de migration de VM vers un conteneur, en transformant les machines virtuelles Linux inactives en proxys sans ressources. SEaMLESS intercepte les nouvelles demandes des utilisateurs lorsque les machines virtuelles sont désactivées, reprenant de manière transparente leur exécution dès que de nouveaux signes d'activité sont détectés. De plus, nous proposons une technique facile à adopter pour désactiver les machines virtuelles basée sur une mise en *swap* de la mémoire de la machine virtuelle. Grâce à notre nouveau système de suspension en *swap*, nous sommes en mesure de libérer la majorité de la mémoire et du processeur occupés par les instances inactives, tout en offrant une reprise rapide du service.

Dans la deuxième partie de la thèse, nous abordons le problème des évolutions à grande échelle des hyperviseurs. Les mises à niveau de l'hyperviseur nécessitent souvent un redémarrage de la machine, forçant les administrateurs du centre informatique à évacuer les hôtes, en déplaçant ailleurs les machines virtuelles pour protéger leur exécution. Cette évacuation est coûteuse, à la fois en termes de transferts réseau et de ressources supplémentaires nécessaires dans le centre informatiques. Pour répondre à ce défi, nous proposons Hy-FiX et Multi-FiX, deux solutions de mise à niveau sur place qui ne consomment pas de ressources externes à l'hôte. Les deux solutions tirent parti d'une migration sans copie des machines virtuelles au sein de l'hôte, préservant leur état d'exécution tout au long de la mise à niveau de l'hyperviseur. Hy-FiX et Multi-FiX réalisent des mises à niveau évolutives, avec un impact limité sur les instances en cours d'exécution.

Mots clés:
Virtualisation de serveurs, hyperviseurs, cloud computing, centres de données, gestion des ressources, migration de machines virtuelles, surcharge de mémoire, machines virtuelles inactives, mises à niveau, mises à niveau du noyau

Abstract

Server virtualization is a technology of prime importance in contemporary data centers. Virtualization provides two key mechanims, virtual instances and migration, that enable the maximization of the resource utilization to decrease the capital expenses in a data center. In this thesis, we identified and studied two contexts where traditional virtual instance migration falls short in providing the optimal tools to utilize at best the resources available in a cluster: *idle virtual machines* and *large-scale hypervisor upgrades*.

Idle virtual machines permanently lock the resources they are assigned only to await incoming user requests. Indeed, while they are most of the time idle, they cannot be shut down, which would release resources for more demanding services. To address this issue, we propose SEaMLESS, a solution that leverages a novel VM-to-container migration that transforms idle Linux virtual machines into resource-less proxies. SEaMLESS intercepts new user requests while virtual machines are disabled, transparently resuming their execution upon new signs of activity. Furthermore, we propose an easy-to-adopt technique to disable virtual machines based on the traditional hypervisor memory swapping. With our novel suspend-to-swap, we are able to release the majority of the memory and CPU seized by the idle instances, yet providing a fast resume.

In the second part of the thesis, we tackle the problem of large-scale upgrades of the hypervisor software. Hypervisor upgrades often require a machine reboot, forcing data center administrators to evacuate the hosts, relocating elsewhere the virtual machines to protect their execution. As this evacuation is costly, both in terms of network transfers and spare resources needed in the data center, hypervisor upgrades hardly scale. We propose Hy-FiX and Multi-FiX, two in-place upgrade that do not consume resources external to the host. Both solutions leverage a zero-copy migration of virtual machines within the host, preserving their execution state across the hypervisor upgrade. Hy-FiX and Multi-FiX achieve scalable upgrades, with only limited impact on the running instances.

Keywords:
Server virtualization, hypervisors, virtual machine monitors, cloud computing, data centers, resource management, virtual machine migration, memory overcommit, idle virtual machines, upgrades, kernel upgrades

Acknowledgments

I would first like to thank my supervisors, Professor Guillaume Urvoy-Keller, and Professor Dino Lopez-Pacheco, for their expertise, patience, and moral support, throughout all Ph.D. Thanks for pushing me beyond what I believed I was capable of. I wish to extend my special thanks to Professor Daniel Hagimont, and Professor Gaël Thomas, for the time dedicated to reviewing this dissertation. I would like to acknowledge all the people on the third floor of the I3S lab. Thanks for every bit of advice, insight, and, most of all, the warm familiar environment you created.

Special thanks to Alessio Pagliari, a colleague, and most importantly a dear friend, who accompanied me throughout this journey. You have been like a brother to me. Thanks also to Adam, Antonia, Melissa, Moudy, and Sara, you kept me afloat whenever I felt I was going down. I will never forget about you. I would like to extend my sincere thanks to all my friend/colleagues in Nice that have directly or indirectly. Thanks for the amazing experience.

Thanks to my family for the encouragement and, most of all, for making me who I am today. This Ph.D. thesis is dedicated to you.

Contents

Chapter 1

Introduction

Contents

1.1 Context and Problem Statement

Instead of a multitude of independently managed servers, the industry opted for the concentration of much of the IT infrastructure in a single well-managed facility, the data center (DC). *Server virtualization* encapsulates heterogeneous computations, services, and applications, into *virtual instances* (VMs or containers), consolidating in peaceful isolation much of the IT workload on the same physical hardware. Companies of any size and kind trust the efficacy and efficiency of server virtualization to the point of embracing the idea of running their core-services in third-party multi-tenant data centers, catalyzing the adoption of *cloud computing*.

Part of the cost saving savings associated with data centers and server virtualization derives from the easier management of virtual instances, as compared to physical machines, increasing the staff productivity and efficiency. Yet another cost saving opportunity, that alone spawned entire new lines of research, involve the concept of *maximizing the resource utilization* in data centers. Under-utilized resources are symptom of an excessive and avoidable purchase of IT equipment, also tied to the waste of the facility physical space—racks, rooms, land—and of electricity to power and cool down unnecessary appliances. The latter is exemplified by the energy inefficiency of commodity x86-64 machines, the prevalent kind in today's data centers, proportionally consuming much more when operating at low utilization rates.

Data centers employ a sophisticated software, called *resource managers*, in charge of planning the placement of VMs and containers on the physical machines. The resource manager juggles with several objective metrics that aim at reducing the operational cost of the data center, while satisfying the

user requirements. Furthermore, as data centers are dynamic, the resource manager needs to constantly improve the placement of virtual instances to meet the evolution of their resource utilization, and to account for events such as the maintenance, failure, decommission of physical machines. Here is where *migration* comes into play. Indeed, a key feature of server virtualization is the capacity of migrating virtual instances from one physical machine to another. The independence between the virtual instances and the underlying hardware (or the OS environment) makes it possible, in principle, to place a VM or a container on every physical machines. Therefore, an already executing instance can be destroyed and re-instantiated to run on another machines. Notoriously, virtual machines are able to keep their run-time state across the migration, resuming transparently on a different physical machine. Sophisticated algorithms can transfer the bulk of the state data in the background, while the instances keep running, reducing the service interruption (*live migration*). Migrations improve the DC resource utilization by adjusting the placement of VMs (*dynamic re-consolidation*), to move instances to safety ahead of host failures and maintenance events [84].

We identified two scenarios where the traditional migrations underlying both dynamic re-consolidation and node evacuation (to face node maintenance), are either ineffective or detrimental to both reduce the wastage of resources and improve utilization. Such scenarios, that we explain in more details below, are: (*i*) *idle VMs in data centers*, locking their physical memory despite their inactivity or very low activity, and (*ii*) *large-scale hypervisor upgrades* that fail to perform rapidly and efficiently under scarcity of resources.

Idle virtual machines in data centers. Recent studies show that idle VMs are a common problem in data centers. For instance, [65] reports that 80% of VMs deployed within a DC are idle, showing little to no sign of user activity. Idle instances are frequently encountered when private companies deploy their own DNS or mail servers [102] or when VMs are used by software developers to design and test new applications, rarely powering them off outside office hours. Even though they are not actively used, these VMs lock the resources they are assigned, *no matter how migration consolidates them.*

Data center operators are left no other choice than *over-booking* resources, to further accommodate new instances. Contrary to the CPU, data center operators avoid to aggressively over-book memory [82] as the consequences of an out-of-memory system unpredictably affects the performance of all the active hosted VMs (swap-in/out, thrashing). This scenario presents an interesting challenge: idle VMs cannot be simply powered off as they usually host network-based services that are essential to end-users. Furthermore, existing solutions either rely on re-engineer the platform (e.g., adopting containers) [104, 102], or ad-hoc application-level proxies that intercept end-user requests

while the instanced are shut down [64]. Consequently, these solutions fail to propose a generic or easily implementable methodology, leaving space for improvements and a novel approach.

Large-scale upgrades of hypervisors. Hypervisors are crucial components in data centers. Any flaw at the hypervisor might affect the performance, security and, availability of the virtual machines running business-critical workloads. Therefore, upgrades must be *quickly* applied over the entire host fleet. However, complex upgrades at the kernel level inevitably require rebooting the hypervisor, thus terminating the running VMs [61].

Live migrations is a strong technique to evacuate the hosts in need of a reboot [84]. Virtual machines are preserved by transparently relocating their execution to healthy hosts, incurring minimal downtime and a limited performance degradation [44]. However, live migration is hard to scale. It consumes data center resources, such as network bandwidth, to sustain the live transfer of the VMs' state. Furthermore, it requires enough host spare capacity to accommodate all the displaced instances [103]. As such, scenarios exist where live migration is undesirable, for instance, in data centers with scarce resources, or when minimal downtime is superfluous (i.e., for fault-tolerant instances such as batch jobs or replicated services). These factors suggest an interesting approach: *in-place upgrades*. In this approach, the hypervisor is fully upgraded without the need to migrate the virtual machines elsewhere. A key challenge is to perform the in-place upgrades without incurring unaffordable VM downtime, offering a valid alternative to the classical live migration approach.

1.2 Contributions

In this thesis we propose two *ad-hoc* migration techniques that overcome, each, the intrinsic limitation of traditional migrations, supplying an additional tool to optimize and ease the management of data center resources.

VM-to-container migration for consolidation in DCs. Idle VMs lead to a waste of resources in data centers, notably memory. Idle VMs cannot be shut down to reclaim their memory because of the service disruption it incurs. A key observation is that VMs contain what we define as *gateway processes*. These gateway processes are a set of user-space processes awaiting for incoming connections (e.g. a web server, or an SSH server) consituting the entry point to the VM. We propose SEaMLESS, a solution that *migrates* the entire set of gateway processes from an idle VM to a resource-less container that provides the service interface to the outside. Idle VMs can therefore be disabled to release their resources. SEaMLESS implements a user-activity detection

mechanism that guarantees the correct and safe execution of the transplanted gateway processes, while running in a resource-less environment that can only sustain limited computations. Upon new activities, the VMs are resumed, and their gateway processes migrated back at their place to transparently respond to user requests. Furthermore, we devised a hybrid virtual machine suspension method, called *suspend-to-swap*, based on the traditional system swap space. This method frees the majority of the memory allocated to and used by a VM, yet providing a fast resume time via the lazy loading of memory pages from the swap. SEaMLESS is designed to operate at the tenant-level. It can increase the resource utilization without the cooperation of the data center or cloud operator.

Across-reboot migration for large-scale hypervisor upgrades. In-place upgrades are appealing as they do not involve resource-intensive live migrations, leading to large-scale hypervisor upgrades. We propose Hy-FiX, an in-place upgrade mechanism designed to apply minor and major upgrades, at the kernel level, to a KVM (Kernel Virtual Machine) hypervisors (e.g., QEMU-KVM). Hy-FiX relies on a hybrid mechanism combining *suspend-to-disk* and *suspend-to-RAM* to checkpoint the small-sized virtual machine hardware state (vCPU, vNICs, etc.) leaving the bulky virtual machine RAM resident in host memory. This technique is paired with a *warm-reboot*, the software procedure to start an upgraded hypervisor without clearing the content of the host DRAM. Hy-FiX implements a *zero-copy migration*, relocating the virtual machines across the reboot, and between two hypervisors without the need to checkpoint their memory, leveraging the data already in the host RAM. Hy-FiX trades off a higher downtime, around 10 seconds, for the high scalability of the in-place upgrades.

Co-located hypervisors for efficient live upgrades. In-place upgrades capable of upgrading kernel-level components inevitably involve warm-rebooting the host and restarting the hypervisor. The warm-reboot not only lasts several seconds but also forces the upgrade procedure to sequentially execute first, the shut-down of the old hypervisor, then, the initialization of the new hypervisor, keeping the VMs unable to run. We propose Multi-FiX, a solution built on-top of Hy-FiX, capable of performing the same class of hypervisor upgrades, with VMs incurring a *network downtime* in the order of tens-of-milliseconds. Multi-FiX replaces the warm-reboot technique that Hy-FiX uses with a *multi-kernel boot*, where a new hypervisor fully is initialized next to the old instance that remains able to execute VMs. Note that both co-located hypervisors run bare-metal without leveraging any form of nested virtualization. The other components of Hy-FiX are adapted to enable the memory sharing between the two co-located hypervisors. As a result, VMs perform a *zero-copy migration*

that exploits the memory sharing between the old and the new hypervisor, incurring the same downtime as live migration.

1.3 Publications

International Journals.

- Segalini, A., Pacheco, D. L., Urvoy-Keller, G., Hermenier, F., and Jacquemart, Q.
  Hy-FiX: Fast In-place Upgrade of KVM Hypervisors.
  IEEE Transactions on Cloud Computing (2021), Early Access

International Conferences.

- Segalini, A., Pacheco, D. Lopez, Jacquemart, Q., Rifai, M., Urvoy-Keller, D., and Dione, M.
  Towards Massive Consolidation in Data Centers with SEaMLESS.
  In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2018), CCGRID '18.

Posters and Abstracts.

- Segalini, A., Pacheco, D. L., Urvoy-Keller, G., Hermenier, F., and Jacquemart, Q.,
  Hy-FiX: Fast In-place Upgrade of KVM Hypervisors.
  In *Proceedings of the ACM Symposium on Cloud Computing* (2019), SoCC '19.

- Pacheco, D. Lopez, Jacquemart, Q., Segalini, A., Rifai, M., Dione, M., and Urvoy-Keller, G.,
  SEaMLESS: A SErvice Migration cLoud Architecture for Energy Saving and Memory releaSing Capabilities.
  In *Proceedings of the ACM Symposium on Cloud Computing* (2017), SoCC '17.

1.4 Thesis Outline

The thesis is organized as follows. In Chapter 2, we introduce the fundamental background to understand the work presented in this manuscript. In Chapter 3, we present SEaMLESS, our solution to the problem of idle VMs in data centers. In Chapter 4, we present our first contribution in solving the problem of large-scale upgrades of hypervisor: Hy-FiX. In Chapter 5, we further investigate the topic of hypervisor upgrades with the second contribution of the subject: Multi-FiX. In Chapter 6 we summarize the contributions of our work and discuss possible future directions related to our contributions.

Chapter 2

Background

Contents

In this Chapter, we present the background for this thesis. Section 2.1 and Section 2.2 briefly introduce the concept of server virtualization, presenting two of the mainstream instances: *hypervisor-based virtualization* and *Operating-System-level virtualization* (OS-level virtualization). Section 2.3 presents the approaches to migrate virtual instances, one of the key benefits delivered by virtualization. Section 2.4 presents the background on *resource management in data center*, while Section 2.5 introduced the approaches to *maintenance in a data center*, with particular focus on on the topic of *hypervisor upgrades*.

2.1 Hypervisor-Based Virtualization

Server virtualization is the practice of partitioning and slicing physical machines, usually high-end—yet commodity—data center machines, into secure and isolated enclosures that offer a familiar environment to IT personnel for

developing and deploying services. As emphasized in Chapter 1, the primary benefit is lowering the capital and operational expenses for purchasing and managing IT equipment, achieved via the consolidation of different IT workloads on the same hardware. There are two approaches to implement server virtualization that operate at the lowest level of abstraction, thus offering the greatest flexibility: hypervisor-based virtualization, and OS-level virtualization. In this Section, we cover hypervisor-based virtualization, whereas, in Section 2.2, we present the basics for OS-level virtualization.

Virtual machines are software entities that implement a virtual version of an entire computer's hardware, including CPUs, memory, I/O devices, and the busses that bridge together all components. In this way, a legacy OS can run inside a VM (called *guest OS*), with unmodified applications hosted on top, offering to DevOps an environment identical to the real-life counterpart.

There are several approaches to implement virtual machines. The most successful one, at least for data center usage, is hypervisor-based virtualization. The hypervisor is the system responsible for running the virtual machines, making sure that the software executing inside the VMs behaves identically as if it were executing on the physical version of that same machine. This task is performed via a level of *indirection* that uses real hardware to implement the effects that the guest software has on the virtual hardware. Since this incurs an execution overhead, the key task of a hypervisor is to achieve the best efficiency. *Direct execution* is the technique leveraged to minimize such an overhead, allowing software inside the VMs to execute on the real hardware every time it is safe to do so. It follows that hypervisors usually constrain the kind of virtual hardware supported for the VMs. For instance, the virtual CPUs must have the same Instruction Set Architecture (ISA) of the physical machine underlying the hypervisor.

2.1.1 Type-1 and Type-2 Hypervisors

Hypervisors are traditionally categorized in *type-1* and *type-2*. Before defining each category, we introduce the role of the *Virtual Machine Manager* (VMM). Although VMM and hypervisor are sometimes used interchangeably, in this thesis we refer to the VMM as the subsystem that virtualizes the CPU and the memory of the VMs [40]. That is to say, it keeps control of the state of the virtual CPUs and the virtual memory, updating it as the VM execution advances. In type-1 hypervisors, the VMM runs, in a so-called bare-metal manner, on the physical machine and directly acquires the physical resources, CPU, memory, and I/O devices that power the virtualization. Notable examples of this category are VMware ESXi [42], Xen [21], and Microsoft Hyper-V [22]. In type-2 hypervisors, the VMM is a client of an OS (named *host OS*). The host OS runs on bare-metal, while the VMM requests the resources to run the VMs. For instance, the VMM may be a user-space process run by the host

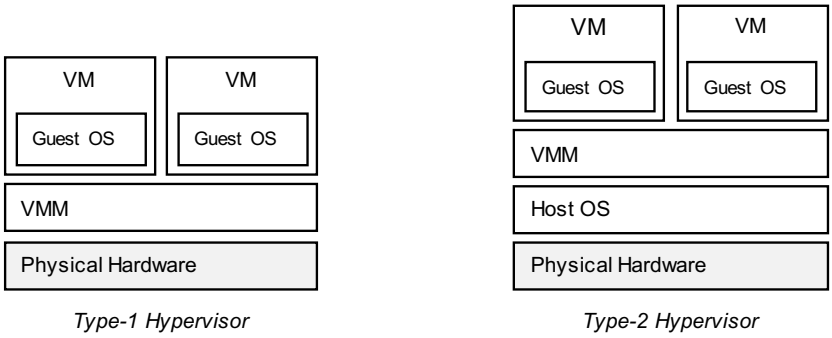


Figure 2.1: Type-1 and type-2 hypervisor architectures.

OS. Such a process gets scheduled from time to time to execute the instructions of a VM on a physical CPU. The VMM may allocate memory via a `mmap` system call (for example on Linux), and perform disk writes via a `write` system call on a file. Notable examples of this category are VMware Workstation [93], and the many hypervisors based on Linux and its *Kernel-based Virtual Machine* (KVM) infrastructure, such as QEMU-KVM, Nutanix's AHV [56], and AWS Nitro [27]. Figure 2.1 depicts a possible stacking of the layers that compose a type-1 and a type-2 hypervisor.

This categorization is important in the context of hypervisor upgrades, a topic of interest of this dissertation. Upgrading an OS, or a hypervisor, requires restarting the whole component, which translates to a host reboot. Type-1 hypervisors can be completely monolithic, hence upgrades at the VMM incur the reboot. Type-2 hypervisors, thanks to their modularity, can have independent components such as the VMM, made replaceable without restarting the whole machine. However, the host OS in type-2 hypervisors plays a fundamental role concerning the stability and efficiency of the hypervisor, hence, frequent upgrades at this level are expected.

2.1.2 CPU Virtualization.

There are three styles to virtualize resources such as the CPU, the memory, and the I/O devices: *full virtualization*, *para-virtualization*, and *hardware virtualization*.

For x86(-64), hardware virtualization was the most recently introduced. However, since the 70s, we regard that approach as the most suitable to build efficient, safe, and faithful hypervisors. Popek and Goldberg first laid out this concept in a theorem. It states as follows. If the set of *sensitive instructions* (with side-effects and/or dependencies to the hardware state) is a subset of the *privileged instructions* (only executable in supervisor/kernel mode), then the CPU can be virtualized via *direct execution* and *trap-and-emulate* [78]. Hence,

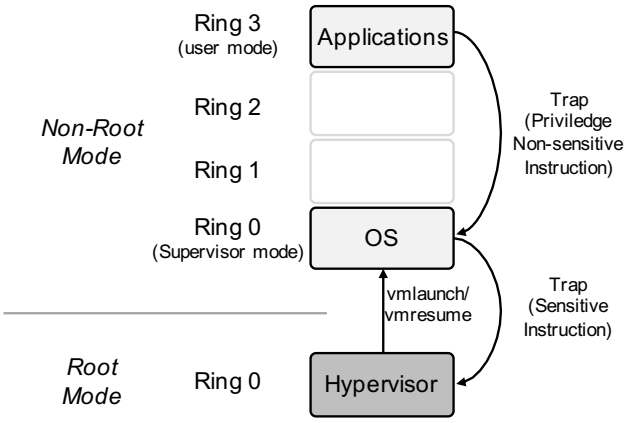


Figure 2.2: VT-x and AMD-V root and non-root mode with respect to legacy x86 protection rings.

the hypervisor lets executing a virtual machine directly on a physical CPU in the least privileged mode, confident that any sensitive instruction that illegally modifies physical machine's hardware state will *trap* because not executed in supervisor mode. The hypervisor re-gains control and *emulates* the effects of the sensitive instruction on the state of the virtual CPU, deceiving the VM that the execution took place correctly. However, x86 does not respect the Popek and Goldberg requirements. This held until 2005/2006 when Intel and AMD introduced their VT-x and AMD-V virtualization extensions for x86 CPUs.

Taking VT-x as a representative, two new modes of execution are added: *root mode* and *non-root mode*. The sensitive instructions of x86 that execute in non-root mode now cause a trap that returns the control to the hypervisor for further emulation. A new set of instructions governs the entering in non-root mode (`vmlaunch`/`vmresume`). Additionally, an in-memory structure, called `VMCS`, stores the state dump of the CPU registers at every transition from non-root mode, letting the VM manipulating its private CPU context. On Linux, KVM is the component responsible for exporting the capability of launching executions in non-root mode. Figure 2.2 depicts the interactions between modes and the protection rings of x86.

Full virtualization and paravirtualization are prior approaches to the release of VT-x/AMD-V. Briefly, full virtualization leverages a technique called dynamic binary translation. Blocks of instructions that the virtual machine is about to execute are analyzed to detect x86 sensitive instructions, which are recompiled, on the fly, with emulated harmless instructions. In paravirtualization, instead of the on-the-fly recompilation, the guest OS code is fully patched before being installed in a VM to replace sensitive instructions with *hypercalls*. The hypercalls are innocuous instructions that cause the VM to trap,

handing the control to the hypervisor to perform the emulation. Dynamic binary translation is heavy on the overheads but manages to run unmodified guest OSes. On the other hand, paravirtualization scores best in performance but requires patching legacy code. Both approaches still play a relevant role in the virtualization of I/O devices.

2.1.3 Memory Virtualization.

Memory is another resource that benefits the x86 virtualization extension provided by VT-x and AMD-V. A virtual machine has an MMU and virtual memory via paging, exactly as real x86 machines. However, virtual machines also have their physical memory space mapped to a virtual memory range in the VMM's memory. To exemplify this, let us consider the QEMU-KVM hypervisor, where the VMM is a user-space process, QEMU, that leverages the KVM APIs to control the hardware virtualization features. QEMU allocates the memory for a VM as a contiguous array of bytes, mapped to its virtual address space. Hence, there are two dimensions of virtual-to-physical mappings. First, virtual addresses within the guest—*Guest-Virtual Addresses* (GVAs)—are mapped to the guest-physical address space, which has a one-to-one correspondence to the array of bytes allocated by the VMM. In turn, the array of bytes in the VMM's virtual memory—*Host-Virtual Addresses* (HVAs)—is mapped to the physical memory of the real machine—*Host-Physical Addresses* (HPAs).

In traditional x86, virtual memory is mapped, page-by-page, to the equivalent chunks of physical memory known as *frames*. The structure that holds the mapping is called *page-table*. The page-table is a hierarchical structure with 4 levels (in x86-64) of subordinate page-tables linked in a tree. The address translation is transparently performed by the MMU, automatically descending the tree (*page-table walk*) to fetch the entry that contains the mapping. As there are multiple virtual address spaces (i.e., every process has its own), the selection of the page-table in charge happens via the `cr3` CPU register, loaded with the physical address of the page-table root. A caching system, called Translation Look-aside Buffer (TLB), is set in place and managed by MMU to avoid excessive page-table walks for recurrent virtual addresses.

The hardware virtualization shipped with VT-x and AMD-V introduces, respectively, the Extended Page Table (EPT) and Nested Page Tables (NPT). The MMU is virtualized inside a VM, so the guest OS can transparently manage its virtual-to-physical mappings via its page-tables set to a virtualized `cr3` register. The hypervisor while in root-mode also sets its own virtual-to-physical mapping, translating the byte array that backs the VM's memory to the HPAs. The EPT combines in hardware the two dimensions of mapping. The GVAs are translated, via the guest page-tables, to the array of bytes in HVAs, conclusively translated to the HPAs via the hypervisor's

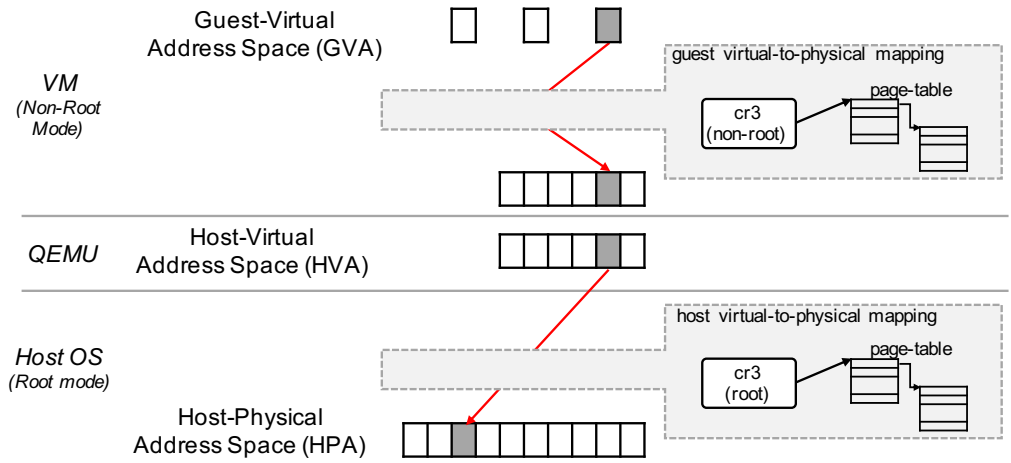


Figure 2.3: Two-dimensional paging in VT-x/AMD-v w/ EPT/NPT.

page-table. Figure 2.3 depicts the relationships between the two-dimensional address spaces in hardware virtualization.

An EPT walk incurs, in the worst case, a number of memory accesses that is squared compared to the single dimension. The TLB assumes a more critical role, and many are the improvements to maximize the cache hits, such as decreasing the number of page-table entries with *hugepages*.

2.1.4 I/O Virtualization.

I/O in modern hypervisors still presents a great variety of approaches. We start with full virtualization, in this context also known as I/O emulation. In such a case, the guest OS runs the unmodified drivers for really existent devices, although virtualized. All the interactions between the guest OS and the I/O, namely *Memory Mapped I/O* (MMIO) and *Port Mapped I/O* (PIO), are set to trap. The effects are emulated by the hypervisor, updating the state of the virtual device to deceive the guest OS into thinking the interaction actually took place. The demanded I/O operations that the virtual device is supposed to perform are transformed into an I/O workload, called *back-end*, that the hypervisor performs on the physical machine's hardware. This approach is regarded as the least efficient due to the emulation of all the hardware details of an existing device, necessary to provide the full compatibility with the legacy guest OS drivers. However, it is a good compromise to virtualize low-throughput high-legacy devices such as the serial port, and the input peripherals (e.g., keyboards, mice).

Paravirtualization has the same trade-off presented for CPU virtualization, sacrificing compatibility for performances. A series of ad-hoc virtual devices, called paravirtualized devices, are paired with ad-hoc drivers deployed into the

guest OS, both designed to offer a virtual I/O that minimizes the required emulation. The paravirtualized devices are simple, implemented as pairs of ring buffers, hence very simple to emulate. Although faster than full virtualization, the drivers for the paravirtualized devices must be ported to the guest OS.

The approaches illustrated so far do not leverage any hardware virtualization. In order to achieve the best performances, the guest OS in a VM would better have the ability to interact with a real device without the hypervisor intervention. This is called *direct device assignment*, or *PCI passthrough*, as it is designed specifically for the class of devices that delivers the highest performance, indeed, PCI and PCI Express (PCIe). We already introduced the list of interactions between the software and the hardware: PIO and MMIO. PIO utilizes dedicated x86 instructions to access the devices' registers, whereas MMIO maps the registers to the physical addresses so they can be read and written via memory operations. Hence, by MMIO-mapping the registers of a real device to the VM's physical address space, the guest OS can interact directly with the device. A complication arises when the guest OS configures a real device to perform Direct Memory Access (DMA) towards the VM's memory. The physical addresses seen by the guest OS do not correspond to the real psychical address seen by the devices. Intel solves this problem within its *Virtualization Technology for Directed I/O* (VT-d), introducing a component called *I/O Memory Management Unit* (IOMMU). Note, AMD's counterpart of this technology is called AMD-Vi. The IOMMU is responsible for translating the physical addresses in the DMA transactions fired by a device. The guest-physical addresses are translated to the HPAs where the RAM is actually mapped. The mapping is specified via page-tables, similar to the counterpart that serves the MMU. We further present more details on the IOMMU and its internals later in Section 5.3.

Direct device assignment grants a single virtual machine the full control of an entire PCI/PCIe device that cannot be shared anymore with other VMs. To increase the scalability of direct device assignment, hardware manufacturers started embedding virtualization capabilities into their cards: *Single Root I/O Vitalization* (SR-IOV). PCIe devices that are SR-IOV capable perform the multiplexing of I/O commands from different VMs in hardware. This feature is exposed to the hypervisor as if a single device is made up of a multitude of independent PCIe endpoints (Virtual Functions, or VFs, in SR-IOV terminology). Each VF can be directly assigned to a VM, and a single SR-IOV card can support thousands of VFs.

2.2 Operating-System-Level Virtualization

Virtual machines provide strong isolation, while also offering a familiar environment to deploy applications. The former is achieved by the legacy guest OS inside the VMs, offering a collection of typical abstractions, such as the system calls, the file-system, libraries, Inter-Process Communication (IPC) objects, and so on. Nonetheless, much of the overhead that virtualization incurs is due to executing a legacy guest OS as an underprivileged entity. Running a great variety of OSes was never the goal of server virtualization in data centers (few mainstream ones are easier to maintain), instead, the focus is on isolation and a deployment-friendly environment. Thus, the virtualization community worked towards an alternative approach: operating-system-level virtualization.

OS-level virtualization isolates groups of processes, including the OS resources they see and access, hence forming what are known as containers. Modern OSes already provide CPU and memory isolation for processes. However, processes still share many *namespaces*, that is to say, they share elements such as the process IDs (PIDs), their position in the global process tree, the mounted file systems, the network stack (IP addresses, transport ports, routing table, etc.), IPC objects, the host name, and the system users. Isolating the aforementioned namespaces (*namespace isolation*) is the feature that container engines such as Docker and LXC leverage to create containers. Control groups (`cgroup`) in Linux, or Job Objects in Windows [95], limit and account for the usage of resources (CPU, memory, disk I/O, network, etc.) for a group of processes, complementing the container isolation. Further security is achieved leveraging features such as `seccomp`, in Linux, to filter the system calls invoked from the container, reducing the surface for privilege escalation attacks.

Containers are generally considered faster than virtual machines, although the CPU and the memory performance are close to hardware-virtualization of VMs [89]. I/O is faster in containers than VMs' fully virtualized or paravirtualized I/O devices, but similar to I/O with directly assigned devices. The memory footprint of a container is in general smaller than a VM, due to the fully-fledged guest OS running within the former [71]. On the other hand, sharing a single kernel does not isolate the co-located containers from OS failures. Furthermore, the larger attack surface, in terms of the number of interfaces that enforce the isolation, makes containers less secure in multi-tenant data centers.

2.3 Migrations of Virtual Instances

Server virtualization decouples the virtual instances from the underlying server, i.e., the hardware for virtual machines and the OS environment for containers. Therefore, server virtualization enables the re-creation of the same virtual instance on every other server in the data center. This principle is at the basis of the virtual instance migration, defined as the actions to relocate the execution of a virtual instance from one machine to another. In this Section, we provide an overview of the migrations of virtual machines, a classical feature of hypervisor-based virtualization. We also introduce the pioneering approaches to container migration.

2.3.1 Virtual Machine Migration

Virtual machines run on virtual hardware. The software inside the VMs, i.e., guest OS and applications, only interfaces with the state of the virtual hardware. Such a state is entirely represented in software within the hypervisor. This property enables the *checkpointing* and *migration* of virtual machines.

First of all, any virtual machine can be paused, known as *suspend-to-RAM*, by simply freezing the hypervisor threads that are executing the VM's instructions on the physical CPUs. The execution state, including the CPU context, remains readily in the hypervisor memory. Hence, VMs can instantly resume as soon as the threads are unfrozen.

Suspend-to-RAM is the cornerstone of another virtualization feature: *checkpointing*. The hypervisor, after pausing a VM, can checkpoint all the information that describes the VM execution state at that moment. This information includes the content of the VM's RAM, the virtual CPU context, the registers of the virtual devices. This technique is known as *suspend-to-disk* when the destination for the checkpointed state is the storage. Note that performing and reversing the suspend-to-disk (*restoring*) costs time, especially checkpointing/restoring the VM's RAM holding gigabytes of data.

VM Termination/Restart. It is seemingly far-fetched to define as migration the simple termination/restart of a virtual machine. However, this approach is effectively adopted in dense data centers, such as public clouds, to relocate instances among different machines. For instance, Amazon EC2 and Google Compute Engine (GCE) leverage this approach to preempt spot instances upon a resource shortage or an outbid of the instance market price [2, 6]. Furthermore, Amazon EC2 adopts it to empty the physical machines in need of maintenance (e.g., hypervisor upgrade) [1].

As the name implies, virtual machines are shut down and terminated from running on a physical machine. Subsequently, another virtual machine is re-created from the same template (machine type, boot image, and other

properties) and booted. This approach assumes a certain level of automation, as the software deployed on the restarted VM shall re-initialize automatically at boot. The execution state is, however, inevitably lost. Stateful services may incur a long downtime, notably if the applications need to load a large amount of data (e.g., in-memory databases [52]).

Cold Migration. Suspend-to-disk dumps the state of a virtual machine to storage. However, if the destination of the checkpointed state is another hypervisor, the virtual machine can migrate and preserve its execution state. We refer to this technique as *cold migration*.

Cold migration is capable of preserving the execution state of a migrated VM, however, the instance remain paused for a noticeable amount of time. The virtual machine can resume only when the entirety of its state is transferred to the destination over the network. The state size of the virtual CPUs, the virtual network cards, the virtual disk controller is only in the order of megabytes, whereas, the content of the virtual RAM dominates with sizes in the order of gigabytes [87]. Assuming a network bandwidth of 1, 10, or 40 Gbps, VMs equipped with at least 5 GB of RAM takes at least one second to be transferred, and so is their downtime.

Live Migration. VMware (with vMotion), and Clarke et al., independently pioneered the approach to live migrate VMs, that is to say, performing a migration without incurring a noticeable downtime [44].

The design proposed, called *pre-copy live migration*, pushes the content of the RAM beforehand, while the VMs keep running, drastically shortening the amount of data transferred while the VMs are paused. This strategy is successful as, assuming the storage is accessed via Storage Area Network (SAN) or Network Attached Storage (NAS), the state of a virtual machine is dominated by its RAM. Pushing the memory is an iterative process that takes place over several *rounds*, in what is referred to as the *pre-copy stage*. During each round, the memory pages modified by the VM's execution, known as *dirty memory*, are re-sent to the destination. Only when the dirty memory size at a given round falls below a certain threshold the pre-copy stage ends. Subsequently, the *stop-and-copy stage* pauses the VM to transfer the remaining of the state. Once the stop-and-copy stage completes, the VM instantly resumes on the destination. The VM only pauses when the remaining data can be transferred in less than the objective downtime, hence, this schema achieves a fixed downtime, generally, in the order of tens-of-milliseconds [84].

Pre-copy presents, however, a defect. Clarke et. al define as *writable working set* the set of frequently written pages that cannot be pre-copied fast enough before the VM writes them again. The writable working set is transferred during the stop-and-copy phase, and, for certain write-heavy

workloads, this fails to meet a short downtime. The worst case is when a VM dirties the memory at a rate higher than the network throughput that pre-copies the memory. In such a case, the downtime incurred is the same as cold migration [44]. An alternative approach to pre-copy live migration is *post-copy live migration* [57]. This approach starts with the stop-and-copy stage to transfer the VM's state except for the virtual RAM, which is kept on the source. The VM resumes immediately on the destination machine, fetching memory *on-demand* over the network upon first-time access. Eventually, all the memory is eagerly transferred and the migration completes.

Pre-copy live migration gained a lot of success. All mainstream hypervisors support this technique, as well as some major cloud infrastructures such as GCE [4] and Microsoft Azure [84]. On the other hand, post-copy succeeds in live migrating write-heavy VMs with a large writable working set. However, the first-time access to a memory page incurs high latency to remotely fetch it, which can heavily affect the performance of the hosted applications. The two techniques can be used together in a hybrid approach. Although they have not disclosed their threshold, GCE switches automatically to post-copy migration when a VMs fails to pre-copy within the target downtime [84].

Limitation of Checkpointing and Migrations. Checkpointing and migrations leverage the property that the state of a virtual machine is completely encapsulated in software. This property holds for the CPU, as its context is exported by Intel VT-x via the `VMCS` structure (AMD-V is analogous). It is in fact especially true for fully emulated or paravirtualized devices, implemented by the hypervisor in software. However, this is not the case for directly attached devices. For instance, some device registers might not be readable, hence impossible to checkpoint [100]. Similarly, other registers cannot be written without producing side-effects, such as triggering a network packet transmission in a Network Interface Controller (NIC) [76]. If the internal device state cannot migrate alongside the VM, the guest driver will likely crash due to the mismatch of states at the destination [76].

One approach is via *hardware state migration*, which however requires the compliance of the device drivers, with device-specific modifications to support it [76]. However, no mainstream hypervisor or platform supports it. Another approach consists in migrating/checkpointing the VM *without any directly assigned device* [100, 85]. This technique leverages the hypervisor and guest OS support for PCI hot-plugging/hot-unplugging. The directly assigned device is removed before the migration/checkpoint, to be later re-inserted after the VM is resumed. In the case of live migration, to maintain the network availability during the iterative pre-copy stage, the traffic is re-routed through a paravirtualized NIC. The directly attached device and the paravirtualized NIC must be enslaved into a single *bonded interface*, which logically aggregates multiple

network interfaces to provide transparent link-failure tolerance (active-backup mode), and higher throughput (load-balancing) [12]. This strategy is, nevertheless, non-transparent to the guest OS. Not only the guest OS must support PCI/PCIe hot-plugging, but the VM user must deploy an automated configuration script to re-initialize the network stack once the card is re-plugged. This approach is widely supported by mainstream OSes (Linux and Windows), and hypervisors (QEMU-KVM, Hyper-V, VMware), and even officially adopted on Microsoft Azure [85].

2.3.2 Container Migration

Containers are harder to migrate as compared to VMs. The state of a container is not perfectly encapsulated as for VMs, but scattered over several OS tables and structures, holding descriptors such as the process PIDs, opened file descriptors, signals, etc. This makes a container hard to checkpoint/restore, and consequently to migrate [44].

Container migration is a subclass of process migration. Process migration was a hot topic in the context of user-space level Single System Images (SSI) [70]. Nowadays, process migration remains relevant in the context of containers, which are, in the end, a collection of processes. CRIU (Checkpoint/Restore In Userspace) is an open-source state-of-the-art solution for process checkpoint/restore on Linux [3]. Presented in 2011, CRIU have since been integrated, although experimentally, into mainstream container engines such as Docker. CRIU leverages an ensemble of kernel hooks, now available in mainline Linux, to export all the components of a the state of a process. Similarly, CRIU leverages other Linux hooks to perform the restoration of a process, such as writing to "`/proc/sys/kernel/ns_last_pid`" in order to force the assignment of a specific PID. The restoration starts with an empty process that morphs itself (opening files, forking children, etc.) into the checkpointed one. When CRIU is used to checkpoint the entire process tree within a container, the whole container is checkpointed. Container migration simply transfers, cold or live, the checkpointed processes to another machine.

Although container migration solutions exist, no mainstream cloud platform supports them, although few private cloud providers offer it in production [19].

2.4 Resource Management in Data Centers

Data centers offer great opportunities to reduce the operating expenses of the IT infrastructure. One example is the wide adoption of energy-efficient hardware, such as components that support *dynamic voltage and frequency scaling*, or the design of facilities that leverage renewable energy sources, such

as free cooling [63]. Another prominent way is pursuing the maximization of resource utilization, such as computational, storage, and network resources. These resources are made available by a wide range of hardware, from CPUs, disks, and network cards, up to racks, and uninterrupted power suppliers, just to mention a few ones. This hardware needs to be purchased, installed, powered on, cooled, and maintained, leading to a high *capital and operational expenditure*. Therefore, maximizing resource utilization reduces the cost of a data center. Server consolidation is a fundamental technique to improve resource utilization in data centers.

A software called resource manager computes the placement of the virtual instances on the physical servers. One placement objective is to pack the virtual instances on the minimum number of hosts so that more hardware can go in power-saving mode. Concurrently, the resource manager manages the *Service Level Agreements* (SLAs) negotiated with the data center users. The resource manager has to guarantee the agreed performance, availability, and affinity (e.g., placing service replicas on different subnets) of the instances, suffering monetary penalties in case of non-compliance.

2.4.1 Static Consolidation

Static consolidation consists of computing the optimal placement for the virtual instances and never changing it [96]. As new jobs arrive, these are placed on suitable machines ensuring that the requested resources are available. Indeed, static consolidation must avoid any resource contention that can degrade the performance of the running instances.

Virtual instances are often *over-provisioned* concerning resource usage [37], severely limiting the consolidation potential of a data center. This phenomenon is partially due to pre-determined instance sizes, similar to what is offered in public clouds [56], but also due to sizing resource demands to face peak loads, which may rarely occur. Consequently, it is necessary to *over-book* resources to boost the data center resource utilization.

Over-booking is the practice of considering available more virtual resources than what is backed by physical resources [89]. The ratio between virtual resources and physical resources is called *over-booking ratio*. Over-booking can lead to a performance degradation due the contention in acquiring a busy resource. We refer to this situation as a *hotspot* [56]. The virtual instance placement shall change according to the evolution of the resource usage, hence *dynamic consolidation*.

2.4.2 Dynamic Consolidation

Dynamic consolidation leverages migration (termination/restart, cold, or live) to correct the placement of virtual instances. Periodically, the resource uti-

lization of the instances is analyzed to monitor and prevent hotspots in an over-booked environment [56]. Dynamic consolidation can also achieve objectives such as reducing the fragmentation of some resources (e.g., making big chunks of memory available for large instance [84, 56]), and evacuating virtual instances to perform host maintenance [62].

Workload monitoring and prediction is a topic closely related to dynamic consolidation. Monitoring the resource usage can be used to dynamically adjust the over-booking ratio, which normally is set static for all machines [38]. A great deal of work focuses on predicting the resource utilization (usage patterns, lifetime) [46]. This enables the informed co-placement of compatible instances, although it is prone to biased predictions, especially for transient or test workloads [38].

2.4.3 Resource Over-booking

We briefly introduced in Section 2.4.1 the concept of *over-booking*, also known as over-commitment, or over-subscriptions. We now characterized how over-booking is adopted in data centers.

All resources in a data center can be over-booked. However, most of the literature focuses on the CPU and the memory as representative of two classes of resources that exhibit different utilization patterns. CPU is characterized by a bursty behavior, with a large gap between average and peak utilization [96], offering great opportunities for over-booking. Besides, the short duration of peak loads translates to low risks of contention. Conversely, memory is problematic. In general, the average memory usage often approaches the peak utilization, resulting in fewer opportunities to over-book [96]. Memory is therefore rarely over-booked, constituting the limiting factor in server consolidation [82, 33, 37, 56]. However, there is a difference between the memory *used* (written at least once), and the *working set*, namely the subset of the memory that needs to be read and written to advance the instance execution. Note that the concept of the WS is related, yet different, to the writable set size of live migration introduced in Section 2.3.1. Indeed, virtual instances that are inactive for prolonged periods, i.e., *idle instances*, have a small working set, *enabling over-booking opportunities*. Around this observation, we developed our project, SEaMLESS, presented in Chapter 3.

The performance penalties due to a CPU or memory hotspot highly differ. For the CPU, the hypervisor/OS multiplexes the CPU-time amongst instances. The CPU context switch is lightweight, incurring a low overhead to multiplex such a resource. On the other hand, both OSes and hypervisors leverage the storage (*swap space*) to multiplex the RAM upon exhaustion of the available space. Thus, the running workloads suffer from higher memory read/write latencies, until the total loss of responsiveness (*thrashing*).

Over-booking for virtual machines is more complex than containers. VMs

are created with a finite amount of resources that cannot easily be extended [89]. Furthermore, some valuable information (e.g., free pages) is not transparently available at the hypervisor to operate the efficient eviction/preemption of the resources [26, 80]. We now detail the traditional over-booking mechanism that hypervisors employ.

Hypervisor Swapping Swapping is a technique to multiplex the RAM upon free memory exhaustion. Only the working set of the running applications/instances requires to be present in memory to allow the execution to advance. if no page is available, some must be evicted to storage to reclaim free RAM. This operation is referred to as *swapping-out*. Pages that are unlikely to be accessed soon are the best candidate for eviction (e.g., approximated by the Least Recently Used—LRU—pages [94]). Note, if a swap-out page is accessed this must be first loaded into RAM. This operation is referred to as *swapping-in*. The swapping-out to reclaim RAM, and the swapping-in to restore a page in RAM, increase the access latency of a page by several orders of magnitude. When the cumulative working set size for all the applications/instances is greater than the RAM, the workloads cannot advance their execution due to the overhead from the continuous swapping of the working set, potentially leading the kernel to stall. This is what we refer to as thrashing.

In the context of hypervisor-based virtualization, two levels of swapping occur. First, the guest OS inside a VM can enable swapping, thus evicting to disk some of its virtual RAM. This technique is of no help for memory overbooking. The reclaimed memory remains within the VM, hence not available externally to other instances. The second level of swapping is performed by the hypervisor, referred to as *hypervisor swapping*, capable of effectively overbooking the memory. Hypervisor swapping bears some shortcomings. The hypervisor, being agnostic towards which memory sits unused within a guest, can inadvertently swap out free memory, leading to a needless and inefficient RAM reclamation. Another similar shortcoming is *double paging* [26]. The guest OS may swap out (or write to storage) pages that are already swapped out by the hypervisor. This results in loading pages from the disk just to write them back to the disk immediately after. These drawbacks are overcome with the co-operation of the guest OS. Notably, *ballooning* enables to reclaim free memory within the guests without the need to leverage the swap, as detailed below.

Hot-plugging The CPUs, the memory, and some I/O device can be hot-plugged and hot-unplugged from the virtual machines, constituting an approach to both provision and reclaim resources. Hot-plugging and un-plugging are non-transparent techniques. The guest OS inside a VM must comply with

the plug/un-plug operation to avoid a potential system crash due to a hardware component suddenly disappearing.

On the one hand, CPU hot-plugging is well supported by mainstream hypervisors such as VMware ESX [20] and KVM [10]. On the other hand, memory hot-unplugging potentially incurs long delays to reclaim RAM, and it is prone to failure. The support for memory hot-unplugging is limited and rare in production environments [26].

Ballooning As said before, the hypervisor has little knowledge of the status of the memory inside a VM. For instance, the hypervisors use *demand paging* to allocate virtual machine's memory, that is to say, only the first-time access to a page prompts the reservation of a physical frame to back such a memory. If an allocated page is later freed by the guest OS, the relative frame stays busy within the hypervisor. As briefly addressed while describing hypervisor swapping, the lack of introspection leads to an inefficient memory reclamation, swapping out pages that hold no meaningful content.

Ballooning is a paravirtualization technique that employs a pseudo-driver installed inside the guest OS that forwards information on which pages are free inside the VM. Typically, a balloon driver works by allocating the unused memory from within the VM (*inflating* the balloon). The allocated memory is kept pinned, preventing the guest OS from ever swapping it out. Ultimately, the balloon driver communicates the set of reclaimed pages to the hypervisor, which recycles such a memory to serve other allocations. Ballooning has another advantage. It raises memory pressure at the guest OS, forcing the former to release non-essential memory such as the file-system cache. The hypervisor can return memory to the VM by *deflating* the balloon, that is to say, the driver frees the memory taken so far.

Ballooning is non-transparent. The guest OS shall comply with ballooning by installing and enabling the relative driver. As a consequence, the virtual machine isolation weakens. In case of over-ballooning, where the hypervisor requests an excessive memory, the guest OS may go *Out-Of-Memory* (OOM), terminating core-application to relieve the pressure.

Ballooning is regarded as an opportunistic approach to reclaim memory [31, 26]. Only free pages within the guest OS are effectively recoverable with no negative consequence. Therefore, ballooning shall remain paired with hypervisor swapping for a certain RAM reclamation in case of a hotspot.

Page Deduplication Page deduplication is another opportunistic technique to reduce the memory consumption of the virtual machines [31]. Page deduplication is based on the sharing of the memory that stores identical content, reclaiming the redundant copies of the same pages. Although not strictly bound to virtualization, page deduplication is prominent to boost server con-

solidation [33, 31]. Virtual machines, especially in cloud environments, are often instantiated from the same templates, making them likely to store identical pages (e.g., holding the same kernel image, or the same libraries).

Page deduplication is operated by the hypervisor that periodically scans the memory allocated for the VMs. When two or more identical pages are found, only a single frame is kept whilst the other copies are freed. All the virtual pages are then mapped to the same host frame and treated as Copy-On-Write (COW). Any new write on any of those pages prompts the re-creation of a copy.

VMware was an early adopter of page deduplication, reporting memory savings as high as 40% across several VMs [33]. However, results in practice differ. The adoption of *Address Space Layout Randomization* (ASLR) and hugepages makes it harder to find identical pages [33]. Furthermore, the opportunistic nature of page deduplication cannot deterministically ensure memory reclamation [31]. Therefore, the solution is secondary to hypervisor swapping.

2.5 Maintenance in Data Centers

Data centers undergo regular maintenance to guarantee service continuity, security, and optimal performances. Maintenance involves upgrading regularly the power and cooling infrastructure, as well as deploying new network fabric and physical machines, and upgrading or replacing single components within the servers [62, 84]. Hardware maintenance always incurs heavy disruption at the software level. Interventions at the power and cooling infrastructure require physical machines to be powered off [91, 84]. Similarly, replacing hardware components within a server (e.g., swapping faulty DIMM, upgrade CPUs) often requires shutting it down first. Therefore, hardware maintenance is usually operated on idle or empty machines. In the context of virtualization, migration (termination/restart, cold, and live) is the preferred tool to evacuate the virtual instances from the affected machines, reducing the disruption caused by these events.

Aside from the hardware, it is of critical importance to maintain the software up to date. The large codebase of services and systems deployed within a data center compels frequent patching and software releases. Security protocols and certifications demand that vulnerabilities are promptly fixed within precise time boundaries (e.g., Payment Card Industry Data Security Standard [23]). Furthermore, many services are constantly enhanced with new features to remain competitive. Google reports that 70% of all instance migrations within GCE are due to software upgrades, against a modest 6% due to hardware maintenance events [84]. We now provide an overview of how software upgrades are tackled in data centers.

2.5.1 Software Upgrades

Software upgrades in their general form require *restarting* the affected components [85]. This operation can be highly disruptive. The software needs to *shut down*, *upgrade*, and *restart* [50]. During these operations, the services provided remain unavailable. Furthermore, data is lost after the shut-down, potentially requiring hours for the software to reconstruct the state [52]. *Rolling upgrades* is a technique to mitigate this disruption but requires the software to support replication. Restarts are performed on small groups of components at a time while their workload is diverted to healthy replicas. Needless to say that this solution does not apply to non-redundant software [50]. Furthermore, completing a fleet-wide upgrade takes time as the groups restart sequentially. [50, 52]. Consequently, a great deal of effort in the past decades was put into developing *live upgrades*.

Live upgrading, also known as hot-patching, and dynamic software upgrading, is a technique to apply upgrades without the need to restart a software component. This approach is the most challenging to generalize due to the *state transfer problem* [50]. Semantic changes to existing global data structures, e.g., adding a new variable, prevents any automated live upgrading tool from correctly transferring the state from the old software to the new one [28]. The burden of solving such a problem is delegated to the developers, dissuading them from adopting live upgrades. However, it exists a class of upgrades that live upgrades are suitable for. Security patches often involve simple modifications at the functions level, e.g., adding a check for buffer boundaries, hence easily applicable with live upgrade tools.

Upgrades based on restarting a software component remain the most general, although disruptive, approach. However, this classic technique can be improved in different ways. The shut-down and restart time can be shortened, and the data/state recovered quicker. This approach requires the software to be equipped with a built-in mechanism of checkpointing/restoring. For instance, Facebook equips their in-memory database software with a fast-start procedure that recovers from shared memory the run-time data left by a precedent version [52]. A prominent class of software in data centers that embeds A highly effective checkpointing/restoring is the hypervisors. In the remainder of the Section, we focus on the classic approaches to upgrade hypervisors.

2.5.2 Hypervisor Upgrades

Hypervisors are complex systems composed of several software layers. In Section 2.1.1 we introduced the typical classification of hypervisors in type-1 and type-2. For type-1, we assume that generic upgrades at kernel-level or VMM-level require replacing the whole hypervisor software. Conversely, type-2 hypervisors benefit from native modularity, with a clean separation

between VMM and kernel. For instance, KVM-QEMU has a well-defined component (QEMU) that performs the device emulation, followed by a set of kernel modules (KVM modules) that virtualize the CPU and memory, and finally the rest of Linux that multiplexes the hardware resources [103]. This separation can be exploited to avoid the costly reboot of the entire host OS. Aside from the VMM, other components are fundamental to run VMs. For instance, virtual switches (e.g., Open vSwitch [11]) implement the network back-ends for the virtual machines. In general, restarting any software that contributes to running the VMs may result in the temporary unavailability (e.g., network unavailable during the virtual switch upgrade), or even the complete loss of the VM execution state (e.g., when restarting the hypervisor).

Hypervisor Live Upgrades. Live upgrade techniques for OSes can be used to patch the host OS in type-2 hypervisors [28, 17, 9]. These tools employ loadable kernel modules to incorporate upgraded function-level patches into the kernel, injecting `jmp` instructions to redirect the execution flow into the new code. Despite the negligible downtime, possible upgrades are limited to simple security patches due to the state transfer problem discussed in Section 2.5.1.

There exists a live upgrade tool specialized in patching the VMM of KVM-based hypervisors: *Orthus* [103]. This tool can replace the emulator (QEMU) and the KVM modules while incurring a VM downtime in the order of tens of milliseconds. Orthus re-architects the KVM modules to incorporate state-transfer capabilities between two consecutive versions, coupled with a mechanism that leverages shared memory to transfer the virtual machine state between two QEMU versions. Being incapable of targeting any complex host OS upgrade, Orthus's main application is patching platforms where the VMs exclusively hold PCI passthrough devices, constraining bugs, flaws, and the attack surface to QEMU and KVM only.

Host Evacuation. Complex upgrades for hypervisors require rebooting the whole host, clearing its state, and resulting in the termination of all the running instances. Thanks to the hypervisor's built-in checkpoint/restore capability, and hence the capacity of migrating virtual machines, a prominent strategy is to relocate elsewhere the VMs. Once the instances are moved to safety, possibly to an already upgraded hypervisor, the empty host can reboot. We refer to this approach as *host evacuation*.

VM termination/restart is the basic migration approach when dealing with replicas or batch-oriented jobs with no high-availability requirements. It fails, however, in achieving transparency, as it does not preserve or recover the VM's state as before the termination. For the same reason, it potentially incurs long downtime due to the instances restarting and warming up their

services [1]. Pre-copy live migration benefits from a negligible downtime but it stresses the network infrastructure while transferring the RAM content of the guests, taking minutes to complete even on fast networks [90]. Furthermore, the downtime for large active instances is proportional to their working set size (Section 2.3.1) leading to a noticeable downtime. With post-copy live migration instances always experience negligible downtime, although at the expense of a substantial run-time overhead due to the remote memory fetching. Whatever is the migration technique, the resource availability in the data center limits the number of hosts that can be evacuated in the same time frame. Due to the high resource utilization sought in data centers, hosts run close to their full capacity. Therefore, the equivalent amount of resources available in the host (CPUs, memory, GPUs, etc.) must be reserved elsewhere to accommodate the displaced VMs. This poor scalability limits the number of simultaneous hosts that can undergo maintenance, and delays the adoption of upgraded software.

Live upgrades and migrations trade off transparency, scalability, short downtime, and upgrade duration, opening for a third way to approach hypervisor upgrades. In this dissertation, we explore the possibility of restart-based upgrades for hypervisors. Instead of leveraging migration to protect the availability of the VMs, we optimize the restart procedure and the state recovery, leading to transparent and lighter upgrades, suitable for large-scale data centers.

Chapter 3

VM-to-container Migration for Consolidation in Data Centers

Contents

3.1 Introduction

In data centers, an abundance of virtual machines remains *idle* due to network services awaiting for incoming connections, or maintaining idle applications with a persistent network session. In public clouds, this happens when user instantiate their DNS, mail servers, and other long-running services subject to a low utilization rate [102]. In private data centers, VMs used by software developers to design and test new applications exhibit frequent idle periods, as they are rarely powered off, even outside of office hours or during holidays. This phenomenon was confirmed in a 2017 report studying the activity levels of the VMs deployed in around 2'000 physical machines from 11 different

facilities [65]. The authors report that only 20% of the VMs shows signs of CPU, network, user, and memory, for more than 5% of the entire observation time, whereas the remainder 80% is idle. Around 30% of the VMs shows no activity at all for over six months.

Idle VMs lead to a *waste of memory* which derives from the limited effectiveness of traditional *memory over-booking* in data centers (as discussed in Section 2.4.3). CPU load rarely peaks, making CPU over-booking ideal to efficiently exploit the time-slices when the resource is under-used. However, unused memory is rare, and memory over-booking translates to the active reclamation of used yet-idle memory (memory outside the working set of the VMs) via techniques such as ballooning, page deduplication, and hypervisor swapping. Ballooning and page deduplication are opportunistic techniques that do not guarantee deterministic memory reclamation, whereas hypervisor swapping suffers the lack of introspection to the guest OS and incurs heavy performance degradation in case of sudden load spikes. Memory over-booking is therefore lightly applied, making it the limiting factor in server consolidation [82, 33, 37, 56].

These idle VMs cannot be simply powered off, as they may host essential service remotely accessed, and new requests may arrive suddenly and unpredictably. Reclaiming the amount of memory allocated to idle VMs is thus necessary. Existing solutions either rely on ad-hoc proxy servers to replace the shut-down VMs while idle [81, 64] or they require radical changes at the platform or hypervisor level [102, 104, 74]. As a result, they fail to propose a generic or easily implementable methodology.

In this Chapter we present SEaMLESS, a framework to replace idle VMs with lightweight *Virtual Network Functions* (VNFs). SEaMLESS *migrates* to a container the state of the *gateway processes*, a collection of processes that make the virtual machine accessible from the outside (i.e., the network). The gateway processes transplanted in the container are the entities that implement the lightweight and resource-less VNF replacing the idle VM. The VNF acts as an application-generic proxy that provides to end-users the feeling of availability while the VM is disabled. The VNF intercepts new end-user requests in order to trigger the transparent restoration of the disabled VM. Furthermore, the VNF can process trivial requests (such as keep-alive messages) that do not require resuming the VM. VMs can be disabled via an array of existing techniques to reclaim the resources they use. Suspend-to-RAM provides a fast VM resuming but reclaims no memory, whereas suspend-to-disk does reclaim memory but incurs in a long restoration delay. Hence, we designed a novel suspension method, called *suspend-to-swap*, to quickly resume a VM upon detection of user activity, while still reclaiming most of the instance memory. SEaMLESS is designed to integrate easily in exisiting cloud platforms. Suspend-to-swap leverages the legacy hypervisor swapping, whereas,

the process that morphs an idle VM into the VNF only leverages user-space tools such as Linux namespaces and Linux process migration, i.e., CRIU (presented in Section 2.3.2). Therefore, SEaMLESS can work at the tenant level, increasing the resource utilization without the compliance of the data center operator (although virtual machine suspension techniques must be available).

Our main contribution is the design and implementation of SEaMLESS. SEaMLESS implements a novel procedure to transform an idle VM (running Linux as guest OS) into a lightweight VNF. SEaMLESS relies on a novel technique to disable VMs, called suspend-to-swap, designed for KVM-based hypervisor which leverages Linux swapping. Moreover, SEaMLESS can be combined with legacy techniques to disable VMs, such as suspend-to-RAM or suspend-to-disk to reclaim the resources locked by the idle instances. We evaluated SEaMLESS and show that:

- Upon user activity detection, the original VM environment is resumed and the services continue their execution transparently with minimum impact on the quality of experience (Section 3.5.2).
- We are able to replace *hundreds* of idle VMs by their corresponding VNFs consolidated onto a *single* physical server or VM (Section 3.5.4).
- Suspend-to-swap readily resumes a suspended VM (Section 3.5.3) while also reclaiming most of an idle VM memory footprint (Section 3.5.6).

In the next Section, we discuss the related work to SEaMLESS (Section 3.2). We briefly discuss the proposals to tackle the problem of idle VMs in data centers, highlighting the differences with regard to SEaMLESS.

3.2 Related Work

In the past, some propositions have been made to tackle the problem of idle VMs in data centers. A bulk of related work on idle virtual machines focuses on energy savings by setting physical servers to low-power modes. Although not active, idle VMs prevent servers from entering deep sleep modes (whenever available) to avoid disrupting the availability of the services deployed inside the virtual machines. Powering off unused servers and increasing server consolidation are two faces of the same coin. Indeed, with memory being the bottleneck to consolidation, reclaiming the memory of idle VMs would consequently lead to energy savings. SEaMLESS can achieve both. Being a framework to replace the execution of an idle VM with a lightweight proxy, the VM can be disabled to reclaim their memory, or, alternatively, consolidated to a server that enters sleep mode.

Partial VM Migration. This family of solutions aims at reducing the energy consumption of a data center by consolidating the working set (WS) of the idle VMs on fewer machines, putting the servers that hold the remainder of the memory on low-power mode. Partial VM migration runs on the assumption that the WS of idle VMs is small and static over time. Partial VM migration is a variant of the traditional post-copy live migration (see Section 2.3.1) without the eager background transfer of pages. *Jettison* is the first proposition leveraging such a technique [36]. Jettison consolidates the WS of idle desktop VMs onto a single server (the consolidation server). The physical machines with the bulk of the VMs' memory are then put in ACPI sleep state S3 (known as suspend-to-RAM[1]). These sleeping physical machines will be resumed as soon as a memory page outside the WS of the partially migrated VMs is accessed (generating a page-fault). The same authors proposed a follow-up work, *Oasis* [104], a solution for idle VMs with generic applications (Jettison only targets physical desktop machines hosting an individual VM). Oasis avoids to wake-up the sleeping physical machines upon the first page-fault by leveraging an ideal hardware state at the physical server that allows to provide the faulting pages while in power-saving mode. The authors do not describe how such a state is implemented. Nitu et al. [74] describe a similar solution, however detailing the implementation of the server low-power mode that can serve remote page-fault. In particular, the authors propose a novel ACPI sleep state, called *Sz*, similar to S3 (PCI/PCIe NIC powered) but with the DRAM actually readable. Nevertheless, such a low-power state is not supported by any manufacturer.

These solutions are transparent with respect to the idle virtual machines. Conversely, SEaMLESS leverages the co-operation of the VMs to precisely identifying the components inside the instances that make them reachable from the outside (i.e., the gateway processes), without requiring estimation of the WS. Also, SEaMLESS directly interposes between a new user request and decides the action to take with respect to the VM, such as resuming the instance on the same server or migrating it elsewhere.

Proxy-based Solutions. In order to transparently disable an idle VM, whether it is suspended-to-RAM, suspended-to-disk, or the whole host put to sleep, an entity shall intercept new user requests, so the disabled VM can resume and the availability of hosted services is preserved. This can be achieved via a proxy intercepting all, or part, of the network directed to the VM. In [81] and [29] the authors employed a sleep-proxy per subnet, to wake up a client machine upon arrival of a network packet for the VM. The solution presented in [81] only filters TCP packets with the SYN flag enabled (new connection) to discard false-positive requests (such as pings) that are not a symptom of

[1]Note, ACPI S3 *suspend-to-RAM* is unrelated to virtual machine's suspend-to-RAM.

new user activity. However, it fails from being generic and ignores wake-up calls triggered by already established, yet idle, TCP connection, or UDP-based applications. *DreamServer* also leverages a proxy custom-tailored for a particular application, co-placing the proxy with a load-balance that intercepts the requests for the disabled VM [64].

SEaMLESS is also based on a proxy, the VNF, which receives the network traffic originally directed to the idle VM. However, SEaMLESS, relies on the gateway process of the idle VM transplanted inside the VNF to faithfully respond to any sort of trivial request such as keep-alive requests. SEaMLESS monitors the interactions between the gateway process and the sink container that encapsulates it, to detect true user activity that requires resuming the VM. Consequently, SEaMLESS transparently supports any protocol (TCP and UDP), and even encrypted channels, such as the one created by SSH.

Platform-based Solutions. In [102], the authors propose *Picocenter* to reclaim memory from idle VMs in a data center. Instead of VMs, constituting bulky entities that carry a high memory overhead (due to the presence of a fully-fledged guest OS), Picocenter leverages customized containers that support the partial *swap-out* of the applications running inside. Picocenter uses a modified version of CRIU, the utility to perform process migration in Linux (introduced in Section 2.3.2), to partially migrate the working set of idle applications inside their containers onto a consolidation server (similar to partial VM migration techniques). Then, Picocenter reclaims the remaining memory by relocating it to the storage (e.g, Amazon S3 in AWS).

Picocenter shares many similarities with SEaMLESS. Both leverage the application itself, kept somehow alive, to maint the network presence of the service. Furthermore, both solutions leverage CRIU to isolate part of the service run-time state that is accessed during the idle execution. Picocenter maintains resident the working set of the applications, whereas SEaMLESS maintains resident the gateway processes that constitute the entry-point to the VM.

The fundamental difference lies in user-activity detection. As for partial VM migration, Picocenter lazily fetches swapped out pages upon faults in the idle application. It leverages the insights given by DNS queries to detect new user requests (resolving the application's hostname) to trigger the total swap-in of the container's memory. SEaMLESS sandboxes the execution of the idle gateway processes, watching the interaction between the processes and the environment to assess if new user activity requires the restoration of the VMs.

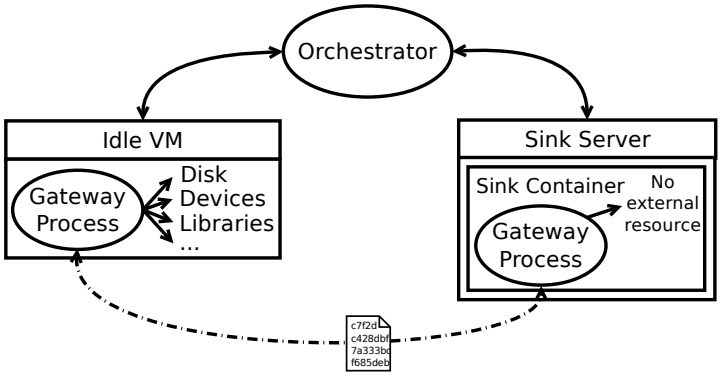


Figure 3.1: Components and architecture of SEaMLESS

3.3 Solving the Idle-VM Problem

Virtual machines in data centers are accessible through several processes waiting for incoming connections or requests by listening to network ports. We refer to each of these processes as *gateway processes*. In this Section, we present how SEaMLESS migrates every gateway process from a VM that has been idle for a long enough time to a *sink container*, constituting the VNF that can replace the idle instance.

In more detail, using Figure 3.1 as an illustration, a virtual machine typically hosts a large set of processes, including gateway processes. Once the VM has been detected as idle, the SEaMLESS *orchestrator*—an agent responsible for the synchronization of the migration—transfers the gateway process to the sink container with all its states and including any open socket, effectively turning it into a virtual network function. Typically, processes access and reference elements such as files, devices, or libraries. These *external resources* are not copied to the sink container to ensure a lightweight environment. When the VNF exhibits signs of user activity, the gateway process will be migrated back from the sink container and restored in its original VM environment to fulfill the user requests.

SEaMLESS is completely transparent from the end-user point of view. Indeed, migrating back and forth the gateway processes between its VM and the sink container is faster than migrating the entire VM, and by keeping the gateway processes running while the VM is paused or turned off, SEaMLESS can maintain any persistent idle connections, either at the transport or at the application layer.

The remainder of this section is organized as follows. Section 3.3.1 focuses on the creation and the structure of the sink container to host the gateway process. Afterward, in Section 3.3.2, we describe the migration procedures to transform an idle VM into a VNF, followed by the technique to detect signs

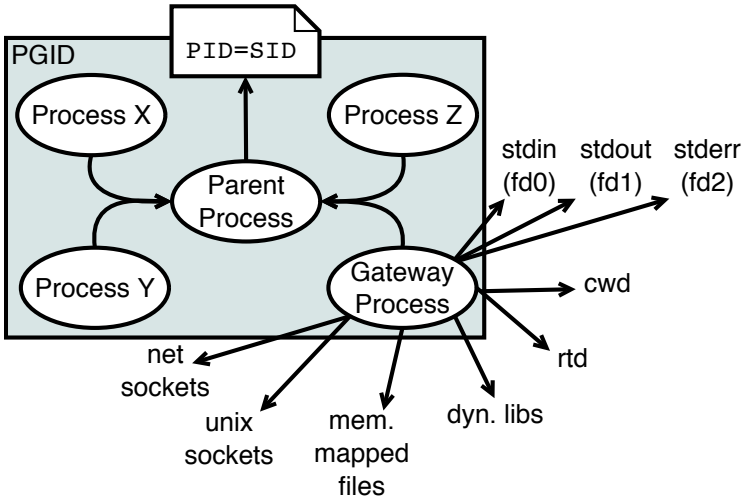


Figure 3.2: A possible gateway process ecosystem and its resources

of user activity at the VNF, which prompts the reverse process to restore the VM.

3.3.1 The Gateway Process VNF

The VNF must behave towards end-users exactly as the idle VM. Therefore, we *migrate* every gateway process from its original environment, inside the virtual machine, to the sink container. The gateway processes must run in an environment that provides isolation while also being lightweight in terms of memory and CPU consumption. For these reasons, we chose to implement the sink container using Linux namespaces, the underlying technology of Linux containers such as Docker (see Sectiont 2.2).

As illustrated in Figure 3.2, when a gateway process runs inside a VM, it is part of a *process ecosystem*, that is formed, among other things, by its process ID (PID), file-system beacons, file descriptors, libraries, etc. Our goal in SEaMLESS is to *only* migrate the gateway process itself, without these side elements. However, this ecosystem is essential for the gateway process to continue executing faultlessly. We now provide some insights into how to achieve this goal.

First, a process is identified by its unique PID. Its value must be preserved to guarantee successful migrations, which we achieve by relying on both the *mount* and the *PID namespace* in our sink container to isolate the `proc` and `sys` filesystems. Consequently, when multiple processes from multiple VMs are migrated to the sink server, these gateway processes can have the same PID.

Second, processes rely on external, dynamically linked shared libraries, which are loaded, unloaded, and linked to a process during runtime. If these libraries are unavailable, the program might experience a segmentation fault. Since VMs deployed in Infrastructure as a Service (IaaS) environments are created from a template image, SEaMLESS can use this very same image to instantiate the sink server, namely the virtual machines hosting all the VNFs from the same template of VM. In particular, it prevents the transfer of the libraries between the VM and the sink server.

Third, a process opens many file descriptors to standard outputs and standard input devices (e.g. `stdout`, `stderr` and `stdin`), but also regular files, directories, device files, and other file-system elements. To handle these *external resources* needed by the gateway process, we decided to point them out to mock *dummy files* in the sink container, expressly created. We leverage the `mount namespace` to ensure consistency between the sink container and the file-system tree inside the original VM. Note, the gateway process is expected to access the external resources *only in case of user activity* (discussed in Section 3.3.3), which triggers the migration of the gateway process back to the original VM. Therefore, the dummy files at the sink container will not introduce any execution error during the gateway process's runtime.

Fourth, a process usually depends on Unix sockets or network-based sockets for communications with the outside (i.e., the network) or other worker processes. SEaMLESS must keep alive existing connections (e.g. established SSH sessions) even after the VM is stopped. End-users must remain connected to the gateway process after the process migration, without connection teardown or data losses. To achieve this, and also to avoid customized kernels with obscure patches, we rely on CRIU (Checkpoint/Restore In Userspace) [3], a modern actively developed software to enable the migration of user-space processes (we already introduced CRIU in Section 2.3.2). The kernel hooks that CRIU leverages, as of Linux version 3.3, are merged into the kernel's mainline [24]. CRIU can *checkpoint* a given process to later *restore* it while preserving all sockets and pipes. SEaMLESS leverages CRIU to dump to disk the state of the gateway processes, and later restore them from their checkpointed image along with all their established network connections (e.g. keeping alive already established SSH session), either at the sink container or the VM.

In summary, we *migrate* a gateway process inside a VM to a sink container. The sink container is composed out of Linux namespaces: a `PID namespace`, a `mount namespace` and a `network namespace` populated with two virtual ethernet devices (`veth`), supported by default in current Linux distributions. One `veth` is connected to the production LAN and must be mirror the network configuration (routing tables, IP address) within the idle VM. The second `veth`, connected to the management LAN, is used to transfer inside the sink containers the gateway process checkpoint image.

3.3.2 Migration Procedures

In this Section, we present the orchestration of the migration of the gateway processes between the sink container and the VM (and vice-versa). Note that the gateway process migration is coordinated with the re-routing of network packets to divert the traffic towards the VNF (and back towards the VM).

Migrating from the VM to the sink server When a virtual machine is detected idle, the orchestrator triggers the creation of the VNF. The migration of the gateway processes inside the sink container follows the steps enumerated below.

Step #1: The orchestrator asks the working VM to dump the state of gateway process to a file. Should any user activity be detected, the VM sends an abort message to the orchestrator to cancel the migration, without any message loss. **Step #2:** Once the checkpointing is done, the working VM sends the state to the sink server. **Step #3:** The sink container restores the gateway process. **Step #4:** The network infrastructure is configured to redirect packets to the sink container. **Step #5:** The orchestrator proceeds in disabling the VM (suspend-to-RAM, etc.).

Migrating from the sink server to the VM The procedure to migrate back the gateway processes is close to the reverse of the procedure described earlier. However, to prevent the long retransmission delay that TCP incurs (TCP Retransmission Timeout) at subsequence attempt to retransmit, we buffer the packets while the gateway processes migrate. The migration of the gateway processes back to the VM follows the steps enumerated below.

Step #1: Upon detection of user activity at the VNF, the orchestrator signal that the corresponding VM must be restored. **Step #2**: Using the Netfilter Queue (NFQUEUE) available in current Linux distributions, a buffer is deployed in the physical server hosting the VM to store packets sent to the VM. At the same time, the network infrastructure is configured to route the packets to the VM instead of the VNF. **Step #3**: The gateway process at the in the sink container is dumped. **Step #4**: Once the VM resumes, the gateway process checkpoint image is uploaded to the VM. **Step #5**: The gateway processes are restored with their updated state (after detection of user activity). **Step #6**: The buffered packets are released from the NFQUEUE, the filter is destroyed, and the communication is now handled by the VM.

Addressing Routing Issues SEaMLESS relies on Software Defined Network (SDN) to reconfigure the network infrastructure into diverting packets from the VM to the gateway processes in the sink container. The orchestrator requests the SDN controller to inject the ad-hoc rules into the SDN switches.

The `veth` interface of the sink container is either configured with the same MAC address VM's interface or the destination MAC address of any packet must be rewritten. If no SDN hardware is available, we assume that data centers providing IaaS support the network operations to reroute the packet in case of VM live migrations. SEaMLESS plugs to the same rerouting infrastructure that legacy live migration leverages to address the relocation of the running instances.

3.3.3 Detecting User Activity

Once the sink container is deployed, the gateway processes inside handle any first-time communication with the end-users. Two questions arise at this point: (*i*) how to detect user activity? (*ii*) How to prevent the VNF from highjacking the full communication with the end-users (which can lead to errors), rather than resuming and handing back gateway processes the original VM?

First, we highlight that not all incoming network packets are a prelude to user activities, such as ICMP, ARP, or application-level keep-alive messages. Such requests must not trigger the resuming of the idle VM. Typically, messages at the transport layer or below are directly replied to by the protocol stack available at the sink container (e.g., ICMP, ARP). However, application-layer keep-alive messages require the execution of the gateway processes. However, we assume such messages are simple enough not to require the fully-fledged context in the VM to be processed. SEaMLESS implements an accurate solution to successfully handle keep-alive applications messages without the need for restoring the VM.

In order to understand how SEaMLESS detects non-trivial end-user activity, we need to understand the behaviors that a gateway process has upon the reception of a request. If the gateway process leverages TCP, a user message might request a new connexion setup with the server, verify the existence of an application-level channel (e.g., SSH keep-alive messages) or ask for external data (not available at the sink container). If the gateway process employs UDP, a user message might carry a membership verification/update at the application level or ask for external data (again, not available at the sink container). After analyzing several gateway processes applications, we have observed that a new TCP connection *returns* the `accept()` system call (or other related syscalls, such as `poll`, `epoll`) invoked by the gateway process on network socket bound to a port.

SEaMLESS creates the sink container with dummy files, as a replacement for all the file-system entities that exist within the original virtual machine context, but are meaningless once transplanted outside. Note, not only file-system entities are subject to this rule. Other aspects of the user-space environment receive the same treatment: IPC objects (UNIX or network sock-

ets), processes (spawning or sending a signal to process: `fork`, `clone`, `kill`), mounted file-system (`mount`, `umount`), and so on. We whitelist a subset of these user-space objects that carry in the sink container a semantically equivalent state to the corresponding object in the VM. For instance, network sockets in the sink container are restored with the same state (IP address, listening port, established connections) as the VM. Therefore, the gateway processes can interact with such whitelisted objects and expect to produce a semantically equivalent result as if they were executed within the VM. We assume that any end-user request that prompts the gateway process to interact with only whitelisted context entities *is trivial and does not trigger the restoration of the original VM*. On the other hand, when a gateway process tries to access a *dummy entity* that does not reflect the state of the original VM, *the execution is promptly stopped, the original VM resumed, and the computation continues within the original environment where it can complete*.

Consequently, to accurately detect end-user activity that prompts the access to a piece of state not exported to the sink container, SEaMLESS relies on *syscalls tracking* over the gateway processes inside the sink container. Syscall tracing can be easily done in Linux using the `ptrace` library, the underlying framework that debuggers, such as the GNU Debugger (GDB), leverage. Technically, with the `ptrace` library, and for each (specified) syscall, the kernel will trap the gateway process (via a `SIGTRAP` signal), and notify SEaMLESS of the syscall. Namely, the gateway process is stopped at every trap. After we have analyzed the recipient of the system call, we can restart the execution of the gateway process by means of the restart (`SIGCONT`) signal, if the SEaMLESS heuristic decides that the sink container can process the message by itself. Otherwise, SEaMLESS stops the gateway process execution completely (with the `SIGSTOP` signal) and starts the migration procedure of the gateway process from the sink container to the VM. Hence, it is the VM and not the sink container that will reply to the user's request.

3.4 Solving the Waste of Memory Problem

SEaMLESS substitutes an idle VM with a lightweight resourceless sink container, implementing a VNF that acts as the idle VM. Consequently, while the gateway processes await incoming user requests, the idle resources owned by the VM can be freed. Note that the idle VM eventually resumes upon new user activity. Hence, the VM's execution state must be preserved while the resources acquired (CPU, memory) are reclaimed.

Most hypervisors support two modes for disabling a VM while preserving its execution state: suspend-to-RAM and suspend-to-disk. We previously introduced such techniques in Section 2.3.1. Suspend-to-RAM freezes the threads that carry the execution of the virtual machine's CPU. The content of

the virtual machine's memory remains resident in the physical host. No memory is released, but resuming the VM only takes few milliseconds. Suspend-to-disk checkpoints the whole state to non-volatile memory. Consequently, the delay when restoring a VM is longer than the equivalent for suspend-to-RAM and proportional to the size of the checkpointed VM's memory. The resume-delay depends on the non-volatile medium where the checkpointed state is stored (HDD, SSD, NVMe), but also the procedure employed by the hypervisor. For instance, QEMU-KVM *eagerly* loads the entire VM's memory before restarting the threads. VMware ESXi adopts a *lazy-restore* approach. The threads are started with the memory not fully resident in RAM, generating page-fault which prompt the hypervisor to load, on-demand, the missing page [101].

Inspired by the lazy restore approach of VMware, we devised a solution, called *suspend-to-swap*, able to reclaim most of the memory used by an idle VM *while* providing fast restoration. Suspend-to-swap is designed for the QEMU-KVM hypervisor. Suspend-to-swap meets the objectives of SEaMLESS: avoid any modification at the hypervisor/kernel, leveraging only available Linux features for easy integration in existing platforms.

Our suspend-to-swap combines *ballooning* and *hypervisor swapping*, available in all mainstream hypervisor solutions. It also leverages Linux `cgroups` to eagerly force the swapping out of the VM's memory. Suspend-to-swap works as follows. (*i*) To deallocate memory from an idle VM, we first inflate the balloon inside the VM to recover the unused memory inside the guest. Note, inflating the balloon beyond the available free memory can lead to a critical memory pressure in the guest OS, possibly triggering out-of-memory errors. (*ii*) Using cgroups in the hypervisor (QEMU-KVM), we limit the maximum memory allowed to a VM to 10 MB. This triggers hypervisor swapping. The hypervisor reclaims the VM memory, swapping-out the memory pages to meet the 10 MB constraint. After this step, the memory footprint of the idle VM is drastically reduced. (*iii*) We remove the cgroup memory constraint on the VM and we perform a *dummy gateway process restoration*. The goal of this dummy restore is to transparently warm (swap-in) the memory pages that stores data needed by the gateway process during the restoration, including some guest OS kernel memory. This step enables a faster response time after resuming the idle VM. (*iv*) At the end of the previous step, a proportion of the memory employed to execute the dummy restoration becomes free again. Therefore, we inflate the balloon once again to fully recover it. (*v*) Finally, the virtual machine is paused (suspend-to-RAM). The execution of the above steps will provide a memory footprint of the idle VM smaller than 600 MB, as shown in Section 3.5.6.

Note that pure eager hypervisor swapping (forced with cgroups memory limits) is not a good option. Hypervisor swapping incurs inefficient memory

reclamation (discussed in Section 2.4.3) as free memory within an idle VM lands to swap. Therefore, it is essential to pair hypervisor swapping with preventive ballooning, to minimize the unsued pages that end in swap-space. After the VM is restored, the balloon completely deflates to return the virtual machines its nominal memory.

Our suspend-to-swap provides fast virtual machine restoration, including the activation of the gateway process, as shown in the evaluation of our wake-up delay in Section 3.5.

3.5 Evaluation

In this Section, we evaluate the performance of SEaMLESS with respect to the perceived end-user Quality of Experience (QoE) and the resulting memory savings. In Section 3.5.2 we evaluate the delay due to the main SEaMLESS components when a gateway process migrates back from the sink container. We assess the impact of our suspend-to-swap strategy in Section 3.5.3. The scalability of SEaMLESS is analyzed in Section 3.5.4, its reactiveness in Section 3.5.5, and finally, we provide insights about the amount of released memory with SEaMLESS in Section 3.5.6.

3.5.1 Network Testbed

We tested our SEaMLESS prototype on one of the clusters of Grid5000 [30], a large-scale testbed for research experiments on distributed systems. We used Dell PowerEdge R430 servers equipped with 2 CPU Intel Xeon E5-2620, 32 GB of memory, 2 Dell PERC H330 HDD in RAID-0, and a 10 Gbps Ethernet NICs.

The testbed consists of three physical machines. The first hosts the orchestrator. The second hosts the sink container. The third hosts the idle VMs. The network infrastructure (where end-user traffic flows), leverages VxLAN tunnels. Rerouting the packets is achieve with OpenFlow (SDN) rules installed in Open vSwitch switches deployed in each host.

3.5.2 Impact on the Quality of Experience

A full restoration process involves the execution of the following phases, each contributing to the unavailability period of the service: (*i*) gateway process state dumping and compression; (*ii*) image transfering; (*iii*) image decompression and gateway process restoration; (*iv*) additional synchronization time between the sink container, the orchestrator, and the VM to dump, transfer and restore a gateway process.

To evaluate the time needed by SEaMLESS to restore a gateway process at the VM, we carried out several tests with different gateway processes on

Application	Main Tasks Time (s)			Total (s)	**Resp. Time** (s)	Image Size (MB)	VNF Size (MB)
	Dump	Transfer	Restore				
Dropbear	0.04	0.12	0.04	0.20	0.41	0.12	11.18
Vsftpd	0.12	0.11	0.05	0.28	0.41	0.11	7.81
OpenSSH	0.13	0.13	0.07	0.33	0.46	0.13	15.93
Lighttpd/PHP	0.16	0.29	0.16	0.61	0.71	0.29	46.43
Apache2/PHP	0.23	0.43	0.24	0.90	0.95	0.43	67.52
Tomcat	0.46	1.17	0.38	2.02	2.16	1.17	206.96

Table 3.1: Response Time of real-world gateway process applications.

their default configuration. The results are available in Table 3.1, where we report the delays due to dumping-compressing, transferring, and decompressing-restoring a gateway process. The column labeled "Total" corresponds to the sum of all previous delays. The response time (labeled "Resp. Time") corresponds to the observed delay between the first packet sent by the client to the VM, and the first packet sent back from the restored machine. The response time comprises the components from the "Total" column, plus the time overhead due to the synchronization of the various events (e.g. the signaling of user activity from the sink container to the orchestrator). The columns "Image Size" and "VNF size" correspond to the compressed image size of a gateway process (in a `tar.lzo` file) and the size of the sink container hosting such a gateway process respectively. Images are securely transferred with the `scp` command, as it would be done in a real data center.

From our results in Table 3.1, we see that the application with the largest image file is Tomcat (1.172 MB), followed by Apache 2 with PHP enabled (0.428 MB). The lightest image corresponds to vsftpd (an FTP/SFTP server) with only 0.107 MB. We would like to point out that the PHP application used with Apache 2 does not impact the sink container size, nor the gateway process image size. Indeed, the gateway process image only includes the main PHP engine libraries, and not the PHP applications themselves, which are loaded as external resources when needed.

The number of SSH worker processes for both Dropbear and OpenSSH (and, therefore, the size of the process image) depends on the number of established SSH sessions. In our tests, we maintained one single SSH connection, resulting to two gateway processes dumped and restored (i.e. the main daemon process, plus the worker process).

From our tests in Table 3.1, we see that the response time of SEaMLESS is generally smaller than 1 second (except for Tomcat). This is lower than the response time typically expected by the end-users, as reported in [43]. We conclude that the delay introduced by SEaMLESS has little impact on the end-user quality of experience.

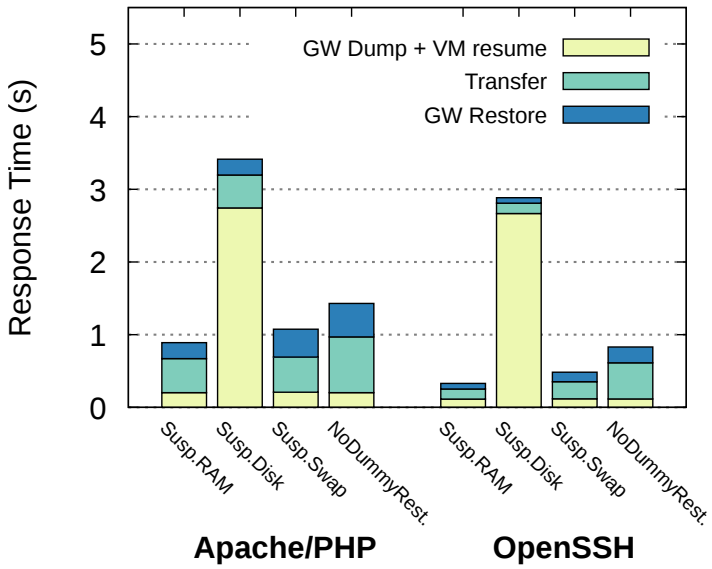
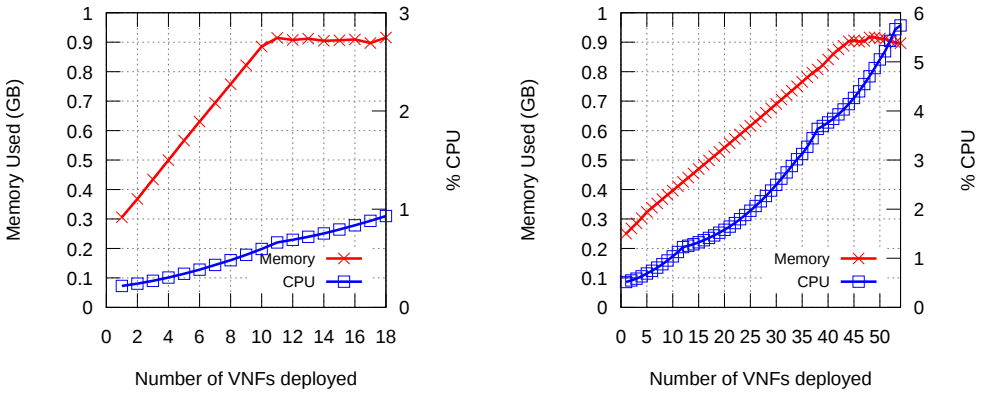


Figure 3.3: Response time when using different disabling techniques on a 3 GB VM.

3.5.3 Impact of Suspend-to-Swap

We discussed several VM disabling strategies in Section 3.3, namely, suspend-to-RAM, suspend-to-disk, and our suspend-to-swap. In Figure 3.3, we quantify the response time of Apache 2-PHP and the OpenSSH gateway processes, with different VM disabling approaches.

From Figure 3.3, we observe that suspend-to-RAM has the fastest response time (around 1 second for Apache 2-PHP and lower than 0.5 seconds for OpenSSH). However, suspend-to-RAM is ineffective to reclaim memory. Suspend-to-disk incurs the worst response time: the time linearly increases with the memory size of the VM, leading to a 3-second response time for a 3 GB instance. Our suspend-to-swap combines the best of both worlds. We show the response time of suspend-to-swap with and without the dummy gateway process restoration. Without the dummy restoration, the resulting response time is around 1.5 seconds for Apache 2-PHP, and 0.9 seconds for OpenSSH. Note, suspend-to-swap actually reclaims most of the VM's memory (details will be discussed in Section 3.5.6). Suspend-to-swap with the dummy gateway process restoration) significantly improves the delay, leading to a response time of around 1 and 0.5 seconds for Apache 2-PHP and OpenSSH respectively.



(a) Apache 2 w/ PHP.

(b) OpenSSH w/ 1 SSH cxn.

Figure 3.4: RAM and CPU used as a function of the number of deployed VNFs.

3.5.4 Scalability of the sink server

In this Section, we focus on the scalability of the sink container in terms of CPU and memory usage. Figures 3.4a and 3.4b illustrate the memory and CPU consumption of the VM hosting the sink container when deploying exclusively Apache 2-PHP VNFs or OpenSSH VNFs.

As expected, CPU and memory usage increase linearly with the number of deployed sink container. The number of VNFs that can be instantiated is limited by the memory: deploying more than 10 Apache 2-PHP VNFs reaches the 1 GB limits (memory usage 100%). On the other hand, CPU utilization remains below 1%. OpenSSH VNFs have a smaller memory footprint. We can instantiate 43 OpenSSH VNFs before saturating the 1GB memory, while the CPU consumption remains lower than 6%.

These results show that a data center server configured with 32GB of RAM can host around 320 Apache 2-PHP sink containers or 1376 OpenSSH sink containers, before experiencing memory swapping. These figures are much higher than the number of idle VMs that could be running simultaneously on the same server with only 32GB of RAM.

3.5.5 Reactiveness

To assess the reactiveness of SEaMLESS, we perform stress tests consisting of the simultaneous resume of a flock of idle VMs. We deploy the VNFs on a single VM with 5 GB of memory and 1 vCPU. The VNFs host OpenSSH and Apache 2-PHP gateway processes. Each test case was executed 20 times.

Figure 3.5 shows the response time (not including the VM resume time), for the simultaneous migration of an increasing number of sink containers. We observe that a linear increase in response time. For one Apache 2-PHP

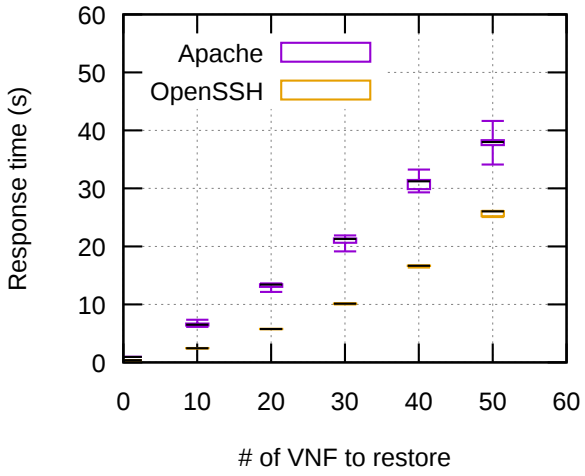


Figure 3.5: Response time as a function of the number of parallel OpenSSH VNFs with user activity.

sink container it is around 0.9 second; 8 seconds for 10 sink containers; and 13 seconds for 20 sink containers. For OpenSSH sink containers, one sink container needs a little less than 0.5 second; around 3 seconds for 10 sink containers; and around 6 seconds for 20 sink containers. Note that the observed response time exhibit little variation, with very narrow interquartile ranges. The disk write/read throughput dominates the response time. The tested host is equipped with a mechanical disk with a much-limited throughput compared to faster SSDs. We advise the usage of SSDs or faster memory supports (e.g., RAM disks) to dump and restore the gateway processes.

3.5.6 Memory Savings

The main objective of SEaMLESS is to reclaim memory from idle VMs. To determine the amount of RAM that can be freed, we need to estimate the expected memory consumption of a memory-reduced VM and the expected memory consumption of its respective sink container. Suspend-to-swap reduces the size of the memory that remains resident in the host as follows. A 1.7 GB VM is reduced to a resident footprint of 474 MB. A 34 GB VM is reduced to about 598 MB. The sink container size depends on the gateway processes deployed. Apache 2-PHP requires around 68 MB, whereas OpenSSH only 16 MB. In Table 3.2, we report the reclaimed memory for typical VM RAM sizes, according to Amazon EC2 instance types [53], assuming the VMs allocate the entirety of their memory.

AWS type	Size (GB)	Reduced Size (GB)	Potential Savings (GB)	Savings %
t1.micro	0.61	0.501	0.109	17.87
c1.medium	1.7	0.474	1.226	72.12
m1.small	1.7	0.474	1.226	72.12
c3.large	3.75	0.496	3.254	86.77
m1.medium	3.75	0.496	3.254	86.77
m3.medium	3.75	0.496	3.254	86.77
c1.xlarge	7	0.515	6.485	92.64
m1.large	7.5	0.511	6.989	93.19
m1.xlarge	15	0.527	14.473	96.49
m2.xlarge	17.1	0.56	16.54	96.73
m2.2xlarge	34.2	0.598	33.602	98.25

Table 3.2: Potential memory savings for common AWS instance types, assuming a VNF size of 67 MB.

3.6 Discussion

There are scenarios in which SEaMLESS is unable to operate due to technical limitations in the underlying technologies (CRIU and `ptrace`). CRIU version 3.5 does not support the reconnection of Unix stream sockets. Indeed, checkpointing and killing a process cause the Unix stream socket peer to close the connection. However, CRIU is constantly in development with a steady monthly release cycle. The developer have the objective of supporting the whole array of IPCs and other file descriptor for a complete checkpoint/restore solution of Linux processes.

In 3.3.1 we stated that instantiating the sink server from the same template of the idle VM avoids transferring libraries loaded by the gateway process. We are aware that any alteration from the original template may result in failures in the restoration. However, different versions are easy to detect, disabling SEaMLESS in such cases.

Since SEaMLESS provides two complementary solutions—one to create a lightweight sink container with the gateway process, and another one to deallocate memory from idle VM—one might ask why the idle VM is not just shrunk, but leaving the gateway process at the VM, without pausing it. Detaching the gateway process (that must always run), from the VM (which can be paused), brings several benefits. First, we have observed that the basic memory set of an idle VM hosting a gateway process slowly grows over time (also observed in [104]), and could even show sudden memory increases due to some services such as application auto-updates. SEaMLESS interposes any new user-activity, avoiding the automatic response from the hypervisor that may result in excessive swapping-in and swapping-out, affecting all the workloads co-located with the resuming idle VMs. SEaMLESS can issue live migration to alleviate potential hotspots following a burst of idle VMs resum-

ing simultaneously.

SEaMLESS is suitable for a selected group of workloads. Monolithic services that leverage shared memory to dispatch the user requests within the application are not supported. SEaMLESS must be able to isolate gateway processes, usually lightweight, from bulky working processes that leverage the resources of the VM (CPU and memory) to execute the request. If the gateway processes and the working processes transparently communicate via shared memory, such interactions cannot be sandboxed by SEaMLESS.

3.7 Summary

In this Chapter, we presented SEaMLESS, a framework to transform and substitute idle VMs to lightweight Virtual Network Functions, allowing such VMs to be disabled in several ways. To achieve such a substitution, SEaMLESS identifies, at the process-level, the portion of VM's state that constitutes the interface from the outside world to the VM: the gateway processes. By leveraging process migration (CRIU), coupled with process sandboxing (ptrace), SEaMLESS enables the *idle* execution of the gateway processes transplanted outside their original VM and placed in a container: the VNF. Monitoring the system-calls through which the gateway processes interact with the enclosing container, SEaMLESS is able to detect non-trivial end-user requests that require resuming of the original VM. Within SEaMLESS, we designed a novel technique, called suspend-to-swap, to disable a VM and reclaim the memory allocated, while also preserving a fast restoration time. Suspend-to-swap leverages a combination of hypervisor swapping and ballooning to proactively evict the virtual machine's memory to storage, reclaiming the precious resource for other usages (e.g., further consolidation).

Our experiments demonstrate that SEaMLESS impacts lightly the quality of experience, thanks to the lazy restoration provided by the suspension-to-swap, reclaiming at the same time most of the memory allocated by the idle instance. Services deployed in disabled VMs via SEaMLESS show a response time of around 1 second for Apache 2, and around 0.5 seconds for OpenSSH; both values including the VM resuming delay and any restoring service procedure. More importantly, SEaMLESS leverages readily available features of mainstream hypervisors and OSes (swapping, ballooning, ptrace sandboxing, containers), requiring minimal effort to be implemented and maintained.

Chapter 4

Across-reboot Migration for Scalable Hypervisor Upgrades

Contents

4.1 Introduction

Upgrading and maintaining up-to-date hypervisor software is a crucial task in data center maintenance, fundamental to improve stability, security, and performance of the virtualized environment and the hosted virtual machines. Upgrading a given hypervisor component (e.g., host OS, VMM) leads to different levels of disruption at the VM level. For instance, in KVM-based hypervisors (e.g., QEMU-KVM) simple vulnerabilities can be transparently fixed with no noticeable downtime with the many Linux live upgrading tools (e.g., *Kpatch*

[9]). Despite the minimal impact over VM availability, live upgrading is limited to patches that replace static code (no changes to data structures) [51], or specific components (i.e., VMM live upgrade in Orthus [103]). Applying complex upgrades to core kernel compartments, e.g., memory management [77], or CPU scheduling [41], require rebooting hosts, terminating VMs, and potentially incurring a long downtime.

The upgrade campaign, ideally, shall be *scalable* and *transparent*, to phase out more quickly the outdated software without affecting the users and their VMs. Considering that kernel patches are released as often as once a month [84], and large cloud infrastructures comprise hundreds of thousands of machines [103], data center operators find themselves in the need of rebooting a large number of hosts. Despite the scale of the infrastructure, upgrades must complete within precise time boundaries regardless of the data center load to comply with security certifications (e.g., credit card security [23]). Furthermore, upgrading hypervisors should neither affect nor involve data center users and their services.

Popular providers trade off scalability and transparency when delivering hypervisor upgrades in their clouds. GCE employs systematic virtual machine *live migrations* to upgrade their hypervisors [4]. Hosts are emptied before the reboot and running VMs get relocated on pre-upgraded healthy servers. Live migration is known to be resource-demanding. This technique consumes high network bandwidth to sustain the live transfer of memory and requires sufficient memory and CPU on spare hosts to accommodate the relocated VMs [37]. Therefore, the number of simultaneous live migrations is limited, resulting in longer evacuation phases and delaying upgrade completion. To highlight the challenge in scaling this technology, [103] reported a 15 day-long delay to upgrade two clusters with 45.000 VMs, due to the waiting for spare capacity, the limited bandwidth (10 Gbps), and operating only in suitable time slots (i.e., at night). Nevertheless, this approach achieves total transparency, avoiding VM termination, and incurring a downtime of only tens of milliseconds [84].

Live migration is not the only approach for hypervisor upgrades, even on platforms such as GCE that heavily leverage it. The cheaper *preemptible instances* [6], meant to terminate in case of resource shortage, do not live migrate. These less critical instances, running batch or fault-tolerant workloads, are terminated and restarted in case of host reboot. Platforms like Amazon EC2 prefer quick, and scalable upgrades, leveraging termination/restart in response to host upgrades [1]. This approach is non-transparent and resets the execution state, forcing users to set up scripts to reinitialize/recover the affected services. Furthermore, a longer downtime is expected because virtual machines not only have to boot, but their services need to warm up, taking up to hours (e.g., in-memory data base servers loading data [52]).

In summary, we observe that the landscape of hypervisor upgrading techniques is rich and not only limited to live migration. In this Chapter, we present Hy-FiX, an *in-place upgrade* mechanism designed to apply arbitrary upgrades to the host OS and other user-space components of a KVM-based hypervisor. Hy-FiX was originally introduced in 2019 in [88]. Hy-FiX relies on a hybrid mechanism of virtual machine *suspend-to-disk* and *suspend-to-RAM* to checkpoint the small-sized virtual hardware state (vCPU, vNICs, etc.), recovering the bulky guest memory directly from the host RAM. Adopting Hy-FiX enables data center operators to promptly issue large-scale upgrades, requiring no external resources to be available. Besides, VMs are transparently preserved, incurring a downtime in the order of seconds, weakly affected by the host memory usage, the number of running instances, and instance workloads.

Our main contribution is the design and implementation of a complete solution to execute quick in-place upgrades for KVM-based hypervisors. Hy-FiX encompasses our novel *memory preserving reboot*, a protocol to start a new upgraded host OS (with upgraded user-space software) while protecting the virtual machines' memory during the transition. Such a protocol is split between the *lazy host memory initialization*, and the *zero-copy memory relinking*, two techniques to quickly start an upgraded hypervisor with the preserved memory made available to the virtual machines. Hy-FiX also adopts a *fast checkpoint/restore* that skips the checkpointing/restoring of the guest memory. This C/R technique, paired with the memory preserving reboot, implements the *zero-copy migration* that quickly transfers the virtual machines' run-time state across the hypervisor reboot.

We evaluated Hy-FiX by benchmarking the upgrade operation on an enterprise-grade host, measuring the VM-level downtime, the performance impact of the lazy memory initialization, and the memory cost to enable the solution. Our key results are summarized below.

- The VM downtime coincides with the hypervisor upgrade time. Running VMs are restored as soon as the new host OS boots, regardless of the CPU/memory load. The upgrade time is weakly affected by the number of running instances and their memory size.
- Hy-FiX lazy host memory initialization reduces the new hypervisor start time from the initial 21.88 seconds down to 7.6 seconds, regardless of host RAM size.
- Virtual machines' performance following the restoration is lightly affected by the lazy host memory restoration and the zero-copy memory relinking. No difference persists in the long run.
- The memory permanently allocated by Hy-FiX to operate is 9 GB for a 1 TB host (0.88%).

The remainder of the Chapter is organized as follows: Section 4.2 presents the related work. Section 4.3 presents the design of the solution. Section 4.4 explains the implementation details. Section 4.5 presents the evaluation results. Section 4.6 discusses limitations and further improvements, and Section 4.7 recaps the contributions of the Chapter.

4.2 Related Work

The classical approaches to software upgrades are (*i*) live upgrades, and (*ii*) the restarting of the component. We presented this distinction in details in Section 2.5.1. Live upgrades are intrinsically limited by the state transfer problem. Even ad-hoc solutions for hypervisors, i.e., Orthus, fall short at live upgrading every hypervisor component. In the presence of complex upgrades, especially targetting the kernel code of type-1 and type-2 hypervisors, data center operators are left with no other choice than to restart the system. Migrations are a prominent tool for both hardware and software maintenance. Migrations empty the host before the hypervisor is restarted, sparing the guests from terminating. However, this approach for upgrades hardly scales in data centers. The limited availability of spare resources and the network bandwidth to sustain the live migrations limit the number of concurrent upgrades, extending the lifetime of outdated software [103, 85].

In-place Upgrades. Scalability is fundamental in data centers with thousands of machines. We refer to *in-place upgrades* as the approaches that strictly operate within the host boundaries, making it possible to upgrade an arbitrarily large portion of the data center simultaneously. Besides live upgrades, in-place upgrades are based on restarting the hypervisor (host reboot). In order to limit the disruption that the reboots incur, both industry and academia have devised schemes to (*i*) shorten the reboot, and (*ii*) quickly checkpoint/restore the run-time state of the virtual machines. The two prominenet families of solutions are: *nested-virtualization-based upgrades*, and *warm-reboot-based upgrades*, the latter comprising our solution Hy-FiX.

In Table 4.1 we compare the whole spectrum of hypervisor upgrades, characterized from the point of view of the scalability (spare capacity, network transfers, upgrade duration), the transparency, the completeness of the upgrade, the run-time overhead, and the downtime incurred at the VM level. In the remainder of the Section, we focus on the in-place upgrades, except for live upgrades already covered in Section 2.5.2.

Nested-virtualization-based Upgrades. Solutions like [67, 48] rely on a thin additional virtualization layer below the hypervisor to enable quick and transparent in-place upgrades. The main hypervisor runs as a guest above the thin

		Spare Capacity	Network Transfers	Upgrade Duration	Transparency	Upgrade Completeness	Run-time Overhead	Downtime
Migration	Termination/Restart (Stateful VM)	✓*	Medium	= Downtime	✗	Full SW + HW	✗	Very high (hours)
	Cold Migration	✓*	High	= Downtime	✓	Full SW + HW	✗	Very high (10s min)
	Live Migration (pre-copy)	✓*	Very high	Very high (10s mins [90])	✓	Full SW + HW	During (long [44])	Very low (10s ms [84])
	Live Migration (post-copy)	✓*	High	Very high (10s mins)	✓	Full SW + HW	During (long [57])	Very low (10s ms [84]))
In-place	Live Upgrade [103]	✗	✗	= Downtime	✓	KVM module, VMM	✗	Very low (10s ms)
	Nested Virtualization [48]	✗	✗	= Downtime	✓	L1 SW**	Permanent	Very low (10s ms)
	Warm-Reboot [87]	✗	✗	= Downtime	✓	Full SW	Post (short)	Medium (13 s)

* Equivalent to the current resource usage on the host to upgrade.
** L1 is the nested hypervisor.

Table 4.1: Hypervisor upgrading techniques summarized. We picture a data center with thousands of large hosts (> 1TB), focusing on large stateful services deployed in VMs.

virtualization layer. The hypervisor to upgrade is shut down and a new one started. However, the two operations happen simultaneously and in parallel: the second hypervisor boots next to the main one, completing its initialization while the old hypervisor keeps running. This strategy erases the downtime due to the hypervisor rebooting.

VMs migrate between the two hypervisors leveraging a *zero-copy migration*, also employed by ou Hy-FiX. The two co-located guest hypervisors share the memory where the content of the virtual machines' RAM is mapped. As a consequence, the VMs' volatile data is immediately available at the destination without the need to copy any bit of it. The upgrade downtime is negligible, although the double virtualization layer introduces an additional overhead [35], which permanently impacts the VMs' performance. We also note that such an approach cannot upgrade the lower-layer hypervisor, which ends up requiring a regular reboot. We cover this family of solutions in more detail in Section 5.2.1.

Warm-reboot-based Upgrades. There are four stages involved in a traditional host reboot: (*i*) hardware reset, (*ii*) firmware (BIOS), (*iii*) boot-loader, and (*iv*) kernel stage [49]. A traditional reboot lasts minutes (see Section 4.5.1) and clears the hardware state of the machine, losing all the run-time state of

the system, applications, and virtual machines. Reducing the duration and disruption that a reboot incurs is crucial to implement a satisfactory upgrade scheme based on restarting the hypervisor.

A Warm-reboot, also known as a soft-reboot, is a technique widely available in mainstream systems (i.e., `kexec` in Linux [59]) that consists of using the running OS as boot-loader to directly start a second OS purely in software. First, the warm-reboot procedure buffers in-memory the OS image, the ramdisk, and other data necessary to boot a new system. Then, the OS shuts down, leaving a single active thread of execution where a `jmp` instruction sets the instruction pointer to executing the new OS image. The warm-reboot skips the first two phases of a traditional reboot (i.e. hardware reset and firmware), and, therefore, significantly reduces the reboot duration. Furthermore, the DRAM avoids the hardware reset, enabling the fast recovery of the execution state left by the terminated OS (i.e., the virtual machine memory content).

Solutions like [79, 47, 66, 69, 98, 61, 106, 85] leverage a warm-reboot to reset or replace an OS, quickly recovering the state from the residual RAM content. *Otherworld* [47] is one of the first applications of a warm-reboot to recover from a system crash. The authors propose the warm-reboot to switch from a crashed Linux instance (kernel panic) to a fresh one previously loaded in memory. Otherworld inspects the memory left by the crashed kernel to recover the list and execution state of the running processes within the previous OS. However, Otherworld does not tackle the problem of service downtime. Indeed, the authors do not discuss the time taken to restore the services.

RootHammer [66] extends the idea from *Otherworld* to refresh a running Xen hypervisor. RootHammer demonstrates that preserving across a warm-reboot the VMs' memory with the virtual-to-physical mapping enables for a quicker restoration than a typical VM suspend-to-disk. RootHammer achieves as such by overhauling the memory boot allocator of Xen, adding a costly procedure to individually mark as protected each page (millions of operations). RootHammer incurs a downtime of 43 seconds on a 12 GB host, demonstrating a poor scalability with respect to the host RAM size.

ReHype [69] derives from Otherworld and RootHammer. ReHype recovers the VMs across a Xen failure. Consequently, ReHype focuses on the detection and repairing of the inconsistencies that the crash incurs. The discussion about how memory preservation is performed and the downtime the VMs incur is missing.

KUP [61] is a solution that leverages a warm-reboot (`kexec`) to replace the Linux kernel while preserving some selected processes. The state of the applications is checkpointed to non-volatile memory thanks to the tool *CRIU* [3] to checkpoint/restore arbitrary Linux processes. The authors propose a re-

covery scheme that leverages memory preservation across the warm-reboot. The old Linux `bootmem` allocator [72] is used to protect certain pages during the early boot stages until the applications are restored. This allocator has been dismissed from the x86 Linux code and must be carefully re-introduced in the current kernel versions. Furthermore, the OS switch time in KUP is 25 seconds and no effective optimization is proposed, especially concerning the growth of host RAM in newer machines.

Our proposition, Hy-FiX, is the first warm-reboot-based upgrade tool for KVM-hypervisors [87]. As for RootHammer and KUP, Hy-FiX preserves the virtual-to-physical mapping for the guest memory, recovering such a memory after the reboot (zero-copy memory re-linking). During the reboot, Hy-FiX protects the guest memory by partitioning of the host RAM in *safe to use during boot* and *memory ranges with data to preserve*. This coarse-grain approach, coupled with the *lazy* memory recovery that defers most of the restoration work when VMs are already resumed, decouples host RAM size from the hypervisor restart duration. Indeed, typical warm-reboots force the VMs to be unavailable for an extended time that linearly increases with the size of the host RAM, a problem that none of the existing solutions tackle.

The same year we presented the full details of Hy-FiX (i.e., 2021), Microsoft published a research paper describing *VM-PHU* [85], a warm-reboot-based upgrade for their hypervisor (a modified version of Hyper-V [25]). VM-PHU is deployed in production in Microsoft Azure. As for RootHammer, KUP, and Hy-FiX, VM-PHU leverage the warm-reboots and the preservation in RAM of the guest memory across the reboot. Notably, VM-PHU tackles the problem of in-flight I/O operations that slow down the fast checkpointing of VMs. Furthermore, VM-PHU is the first warm-reboot-based upgrade solution that supports the transparent checkpoint/restore of *non*-SR-IOV devices directly assigned to the VMs. Hy-FiX handles differently memory preservation. Hy-FiX leverages a bitmap instead of the proposed sorted linked list to record the pages to preserve. This Hy-FiX feature, combined with the lazy restoration, avoids the fragmentation-related problems that affect the downtime of the VM-PHU.

4.3 Hy-FiX: Architecture and Design

Hy-FiX reboots the host to achieve the complete upgrade of the hypervisor. Upgrading a KVM hypervisor consists of replacing the following critical components: (*i*) the host OS and the (*ii*) VMM (which includes the emulator, e.g., QEMU, and the KVM modules). Replacing a component implies restarting it, and for the OS, this translates to a host reboot.

The reboot clears the hypervisor state, resulting in all the running instances being terminated. Hy-FiX objective is to efficiently restore the hyper-

visor state once the reboot completes, bringing back all the VMs precisely as they were before the upgrade. The hypervisor state generally comprises:

- Virtual infrastructure configuration: information on hosted VMs (sizes, number, virtual hardware configuration), local and remote storage, and the virtual network configuration.
- Virtual hardware state: virtual devices such as vCPUs, vNICs, virtual disks, etc., implemented by the VMM as a collection of emulation routines and data structures that hold the context of device registers.
- Virtual machine memory: the content of each VMs' RAM.

The virtual infrastructure configuration is provided by the data center orchestrator (e.g., OpenStack). It is typically uploaded to a local agent running on the host which actuates commands on the orchestrator's behalf. The size in bytes of this state is negligible compared to the other parts of the hypervisor state, hence we assume that the cloud orchestrator offers APIs to restore it right after the reboot.

The virtual hardware state is more complicated. The corresponding data depends on the version of the VMM implementing it. Hy-FiX upgrades the VMM, so the new version may not be compatible with the state within the old hypervisor. Typically, VMMs observe forward compatibility [15], meaning that the checkpoint image of a virtual machine can be supplied to a later VMM version. For Hy-FiX we assume that the VMM, in our case QEMU with KVM [15], supports this feature.

In contrast to the virtual hardware state, the virtual machine memory content depends on neither the VMM version, nor the host OS, and any other piece of software in the hypervisor. *The content of a VM page is the same as the content of the host frame mapping it.*

Hy-FiX implements two techniques to preserve and recover the virtual hardware state of the VMs and their memory content. (*i*) Fast checkpoint/restore, instructs the VMM to serialize the VM hardware state and save it to as non-volatile support. The content of the virtual machine's RAM is skipped. The complementary procedure is executed after the reboot to restore the VMs. (*ii*) Memory preserving reboot protects across the reboot the content of the host frames that map the VM's RAM. We discuss these techniques in detail in the remainder of the Section.

4.3.1 Fast Checkpoint/Restore

The VMM must serialize into a standard format the virtual hardware state for a running VM (e.g., QEMU format [15]) to maintain forward compatibility with an upgraded version. Hy-FiX explicitly instructs the VMM *not to* dump the VM RAM content. The data saved to non-volatile memory by the fast

C/R is in the order of tens of megabytes. The read/write operations lightly impact the whole upgrade downtime. We address this aspect in more detail in Section 4.5.1). The above technique ensures the preservation of the VM virtual hardware state. We next present how to take care of the VM memory.

4.3.2 Memory Preserving Reboot

Host OS Switch Hy-FiX leverages warm-reboots to replace the host OS and all the additional software running on top (i.e., the VMM comprising QEMU). Different from traditional reboots, no hardware reset takes place. The warm-reboot does not clear the host RAM, enabling for *memory preservation across the reboot*. Hy-FiX enriches the warm-reboot protocol by forcing the new host OS to boot within the *safe memory region*.

Despite the absence of a hardware reset, the integrity of virtual machine memory is still at risk. While the new host OS is booting, it may unintentionally allocate and overwrite the memory we are trying to preserve. When Hy-FiX is first enabled, during the host commissioning, it identifies a RAM area called *safe memory region*. Such an area is designated to hold the image and the boot-time memory of the upgraded hypervisors started with the warm-reboots. The frames that map the virtual machines' pages cannot be allocated from the safe memory region. As a consequence, a legacy host OS can reliably boot within that area without the risk of corrupting any preserved memory. Hy-FiX makes sure that the warm-reboot loads and confines the new host OS within the safe memory region, preventing the former from accidentally accessing it.

Lazy Host Memory Initialization Hy-FiX lazy host memory initialization speeds up the host OS boot time. During the boot, the OS spends a significant amount of time initializing the memory. We measured the aforementioned behavior in Linux, in Section 4.5.1. The OS registers the memory regions as usable or protected. For each usable area, the OS executes several per-frame operations that enable the usage of the virtual memory. Furthermore, the OS initializes the memory manager, filling the tables and data structures that track the system's free memory (e.g., Linux buddy allocator free page lists). On machines with hundreds of millions of pages, this prolongs the booting phase proportionally to the size of the total installed memory, lasting more than 11 seconds on our Linux machine with 768 GB of RAM. Hence, we devised a lazy host memory initialization that:

1. Initializes first the safe memory region, which fixed-size is small (e.g., 16 GB). Thanks to the modified warm-reboot, the new host OS stays confined in that region.

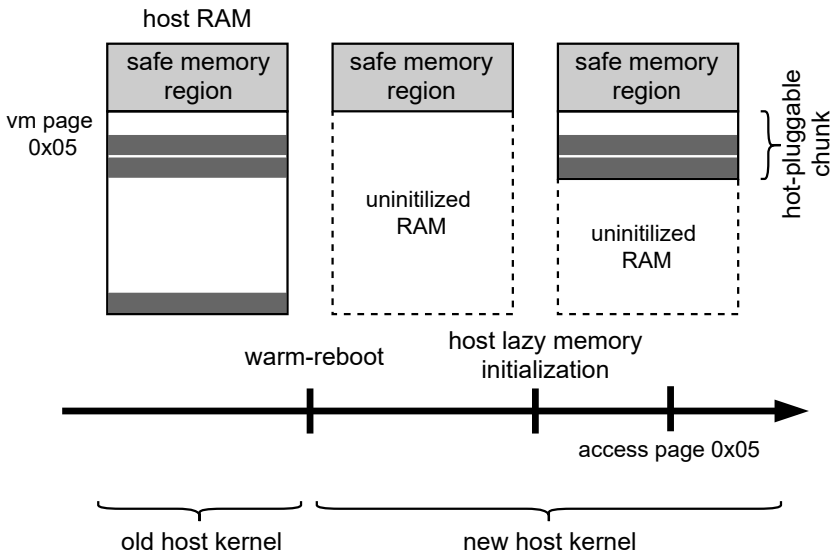


Figure 4.1: Hy-FiX memory preserving reboot.

2. Initializes as a *background task* the remaining of the memory outside the safe region once the OS has fully booted (i.e., the user-space environment is ready).

As a result, Linux full booting time is around 8 seconds (on our target 768 GB machine), independently from the host RAM size, decreasing consequently the VMs' downtime.

VMs restart without the memory being fully initialized. The lazy host memory initialization marks VM memory as non-accessible to intercept the first-time access. The resulting page-fault is handled by *initializing*, on-the-fly, the 128 MB-aligned chunk of physical memory that includes the requested address. The memory page is mapped accessible and the read/write operation resumes. Note that the 128 MB chunk is only initialized once at the first page-fault involving it. Figure 4.1 illustrates the whole host OS switch, performed with the warm-reboot and the lazy memory initialization, achieving together the memory preserving reboot.

Recovering Memory Across Reboots The memory preserving reboot protects the memory content outside the so-called safe memory region, where the virtual machines' RAM is mapped. Preserving such memory is insufficient to allow the resumed VMs to access their memory. The missing piece of information to recover is the virtual-to-physical mapping, i.e., the *page-tables'* content that maps the physical guest addresses to the HPAs (physical addresses within the host). The hypervisor manages the page-tables, which are kept in host

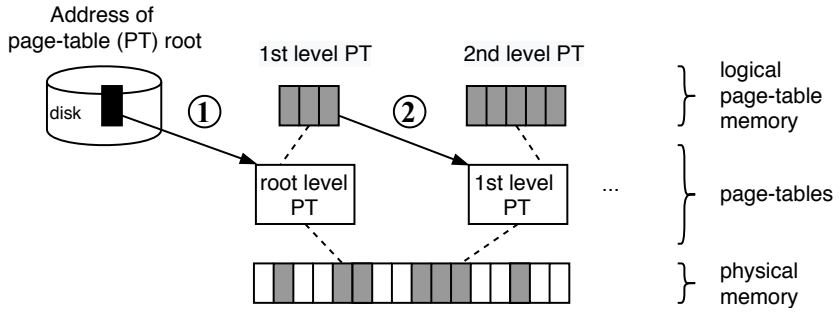


Figure 4.2: The virtual-to-physical mapping for the VMs' memory is recovered from preserved multi-level page-tables. The physical root address of a page-table recursively reveal the location of the whole structure.

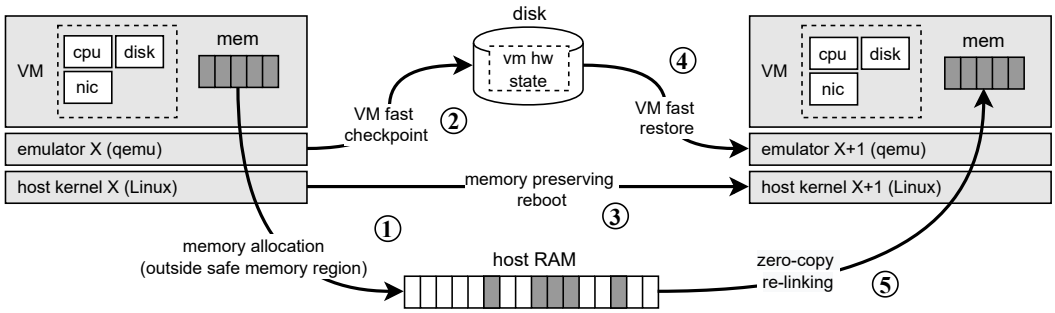


Figure 4.3: Hy-FiX hypervisor upgrade.

memory. As for the content of the virtual machines' memory, the hardware dictates the format of the page-tables. Since their representation is stable with respect to the host OS, and other software layers, page-tables are a perfect candidate to be preserved—as-is—in memory. In Hy-FiX, the page-tables are kept *outside* the safe area region and preserved across the reboot.

We call this technique *zero-copy re-linking*. A single 64-bit integer—the page-table root address—enables the recovery of the whole virtual-to-physical mapping for a VM. Figure 4.2 shows the mechanism in action in a simplified scenario, where page-tables only have two levels (note that x86-64 uses 4 levels). The physical root addresses of every page-tables that maps VM's are dumped to non-volatile memory. Upon restoration, the addresses are loaded and the mapping is restored.

4.3.3 Hy-FiX Upgrade-cycle

Figure 4.3 provides an overview of the whole upgrade cycle with Hy-FiX. Upon the creation of a VM, the memory is allocated outside the safe memory region that the new host OS uses to boot ①. When an upgrade is issued, the binaries (i.e., the new OS image and the user-space binaries) are downloaded locally on

What	Version	# Files	LoC Modified	LoC New
Linux	5.2	8	297	-
Linux Modules	5.2	2	-	1431
kexec-tools	2.0	2	22	-
QEMU	2.11.0	4	48	-

Table 4.2: Hy-FiX code analysis.

the node. The new OS image is buffered in the host memory and stands ready to be later copied by the warm-reboot procedure into the designated location (within the safe memory region). Fast C/R is invoked to checkpoint the virtual hardware state of every running VMs (their memory remains resident in host RAM) ②. Fast C/R pauses the VMs, and the unavailability starts. The warm-reboot switches the host OS effectively booting the new hypervisor ③. The upgraded hypervisor fully boots and initializes the user-space with the VMM. The VMs are restored from the checkpointed image ④, the virtual memory space is restored via the recovered page-tables ⑤. The virtual machines are fully restored becoming available again. The physical memory storing the VMs' RAM content is lazily initialized within the host OS upon the first access.

4.4 Implementation

A proof-of-concept for Hy-FiX has been implemented for the QEMU-KVM hypervisor on Intel x86-64. The PoC is based on QEMU version 2.11, and Linux kernel version 5.2. We report in Table 4.2 the total modified/added code to legacy components (Linux kernel, kexec-tools, and QEMU). The majority of new code resides in two Linux loadable modules for a total of 1431 lines of code (LoC). The two modules implement the lazy host memory initialization and the zero-copy memory re-linking (Section 4.3.2). The single modification to the Linux kernel consists of exposing the ad-hoc routines that execute the individual steps of the Linux memory *hot-plugging*. We leverage such routines to implement the lazy initialization of the memory.

4.4.1 Host OS Switch

Hy-FiX leverages `kexec` [59], an upstream Linux feature that implements the warm-reboot procedure. The memory preservation technique presented in Section 4.3.2 is implemented by customizing the physical memory layout map (`memmap`), which describes to a booting Linux the physical address ranges corresponding to the usable RAM or the reserved regions destined to other uses (e.g., mapping hardware registers). This `memmap` is conventionally populated

by the boot-loader and stored inside the *zero page* [18], the structure that also holds the Linux boot-time parameters. Hy-FiX populates the `memmap` so that the memory to protect, outside the safe region, is hidden and not presented to the booting OS. The booting OS, not aware of such a memory, cannot inadvertently use it. Conversely, the safe memory region will be the only usable RAM region presented to the booting kernel.

4.4.2 Fast Checkpoint/Restore

Hy-FiX leverages the QEMU save/load functions to serialize/deserialize the virtual hardware state of running VMs. QEMU is patched to skip the dumping of the memory [68]. The patch prevents QEMU from checkpointing any virtual machine's memory marked as shared. However, since the release of QEMU 4.0.0, such a feature has become upstream under the migration flag `x-ignored-shared`.

In Hy-FiX, the memory recovery is performed by two kernel modules that together implement an agent called Reboot-Persistent Memory Manager (RPMM). The RPMM loads the page-table root addresses from the storage to re-build the virtual address space for each virtual machine. The reconstructed memory is exposed to the VMM (QEMU) as a set of memory-mappable (`mmap`) *virtual device file*, one per VM. We then leverage an option of QEMU to back the guest memory with a memory-mappable file, providing the virtual device files exposed by the RPMM. Note, is a classic way in QEMU-KVM to provide advance features to the guest memory. For instance, `mmap`-ing a virtual device file is also used to provision *hugepages*, or to share the VM's memory with other host applications (e.g., `vhost-user` protocol).

The RPMM adopts the same interface via virtual device files to expose memory allocation for new VMs. In order to create a new VM, the RPMM is notified to create a virtual device file that QEMU `mmaps` to back the guest RAM. RPMM servers the requested memory by allocating frames outside the safe memory region. This also allows the RPMM to be in control of the page-table that hold the virtual-to-physical mapping for the VM.

4.4.3 Recovering Memory Across Reboots

The RPMM module manages the memory that survives the warm-reboot, providing mapping reconstruction, and the allocation of memory outside the safe memory region. As the Linux page allocator disallows to chose the location of the frames that serves an allocation, the RPMM iteratively requests frames until a suitable one is found. Given the small size of the safe memory region, only a few iterations are needed to obtain a frame outside this area. Once a frame is allocated for a VM, the RPMM records the information in a bitmap, the *preserved memory bitmap*, describing the status of all the frames

in the host (one bit per frame). After the reboot, the bitmap indicates to Hy-FiX which frames were mapping VM's memory within the old hypervisor. Consequently, those frames are kept reserved and not returned to the page allocator upon initialization of the memory. The bitmap resides inside a single huge-page (e.g., 1 GB is enough for a 32 TB hosts), at a known physical address.

In contrast to what we presented in Section 4.3.2, our current PoC does not preserve the entire page-tables in host RAM. To simplify the implementation, we partially dump to storage the second-level page-table entries (PMD in Linux [73]). Note, we checkpoint the second-level entries outside the critical path when VMs are still running. This barely affects the experiments. Even for a large RAM, the memory footprint of the second-level entries is modest (less than 1 MB for a 256 GB host). The time to load such data from disk is negligible compared to the magnitude of the upgrade duration (tens of milliseconds according to Figure 4.6).

4.4.4 Lazy Host Memory Initialization

Linux supports memory hot-plugging for the x86-64 architecture [86]. Hot-plugging is based on the *SPARSEMEM* model [14]. SPARSEMEM logically divides memory into chunks, called sections, of 128 MB. The Linux kernel populates an array with `struct page` entries for each frame within a section. Such a structure is called `mem_map` (not to be confused with the physical memory layout `memmap` mentioned in Section 4.4.1). Linux implements hot-plugging in three phases:

1. Physical memory addition. When a new DRAM module is physically added to the motherboard, the firmware notifies the kernel to register the physical addresses where such RAM is mapped.

2. `mem_map` population. For each frame within the added memory, entries of the `mem_map` array are created and populated.

3. Logical memory addition. The added frames are handed to the page allocator, hence added to the free-page lists that are later used to serve memory allocations.

Phases (2) and (3) can take a significant amount of time on large hosts. If executed at boot-time, the two phases are responsible for a long system start-up time (as shown in Figure 4.7). In Hy-FiX lazy memory initialization, we execute phases (2) and (3), on-demand, upon the first-time access to a page within the 128 MB section. Hy-FiX defers this memory initialization cost only when the memory is needed. We modified the Linux hot-plugging subsystem to expose as internal kernel APIs the functions responsible for phases (2) and

CPU	2 x Intel Xeon E5-2680 (2.40GHz, 14 cores)
RAM	768 GB
Storage	SSD SAS PX04SMB040
Network	Intel 82599ES 10 Gbps

Table 4.3: Testbed node specifications.

(3). The RPMM module invokes these functions to lazily initialize a target 128 MB section.

4.5 Evaluation

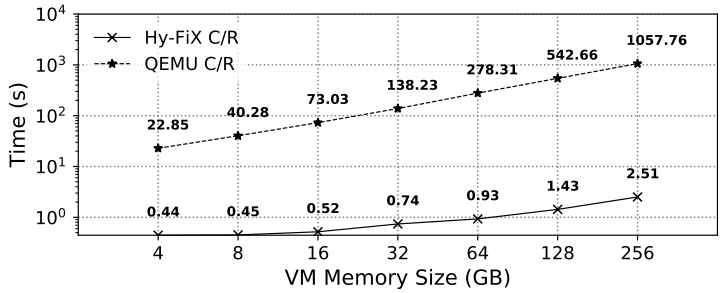
In this Section, we present the evaluation of Hy-FiX along several dimensions. First, we benchmark the time and scalability of the fundamental tasks that compose an Hy-FiX upgrade: the memory preserving reboot and the fast C/R. Second, we analyze the impact that lazy memory initialization has over the resumed VMs. Third, we analyze the memory overhead that Hy-FiX compels. Fourth, we analyze and discuss the end-to-end upgrade time and downtime for different configurations of running instances. The experiments have been conducted on Grid5000 [30], on a host whose characteristics are reported in Table 4.3.

4.5.1 Micro Benchmarks

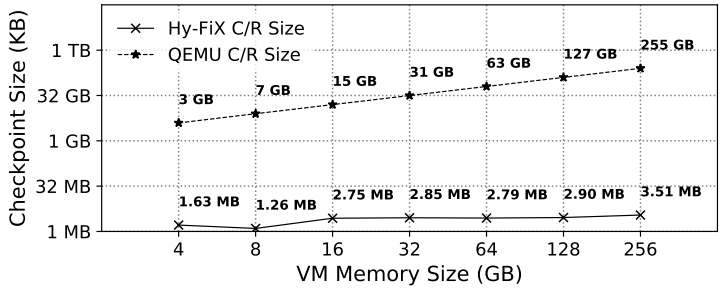
Fast C/R Analysis We compare the total time to checkpoint/restore a single VM instance for Hy-FiX fast C/R and the QEMU original suspend-to-disk. The target VM is equipped with a single vCPU, a single vNIC, a disk controller, a CD-ROM drive, and a progressively larger RAM. Furthermore, the VM memory is pre-filled with random data to simulate the activity of a production workload.

Figure 4.4a reports that Fast C/R duration in Hy-FiX for a 4GB VM is 0.44 seconds, 0.2% of legacy QEMU C/R time. For a larger 256 GB VM, Fast C/R duration is 2.51 seconds, 0.02% of QEMU C/R, and 2.07 seconds more than Hy-FiX's for a 4 GB VM. This result proves the effectiveness of skipping the serializing/deserializing of the virtual machine's RAM. Figure 4.4b shows the size of the virtual machine checkpointed state, comprising only the virtual hardware state for a single VM. The size averages between 1.63 MB and 3.51 MB, remaining mostly constant with respect to the guest size.

Fast C/R duration weakly increases with respect to the guest RAM size. However, as Hy-FiX leverages QEMU legacy C/R routines, it inherits some size-dependent tasks within. During the checkpointing, QEMU builds a bitmap mapping all the guest pages, conveying the information of whether



(a) Checkpoint/restore time.



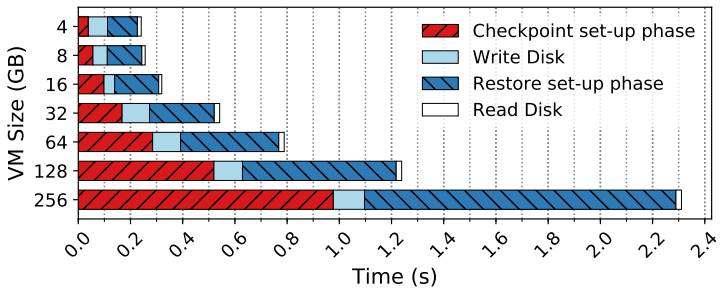
(b) Checkpointed state size.

Figure 4.4: Comparison QEMU checkpoint/restore vs. Hy-FiX checkpoint size for different VM sizes.

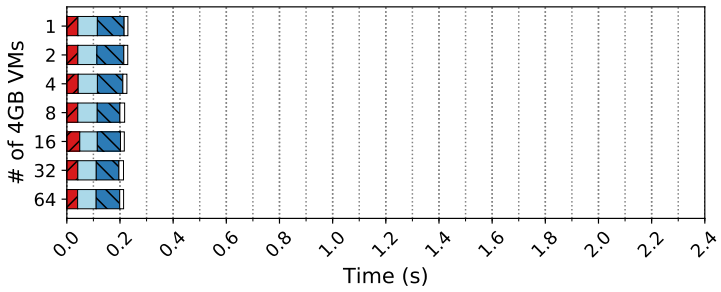
a page shall be dumped or not. We call this operation *checkpoint set-up phase* as it takes place before the serialization of the VM state. Upon restoration, a new QEMU process starts. QEMU registers the allocated virtual machine's memory within KVM. We call this operation *restore set-up phase* as it takes place before the deserialization of the VM state. We isolated the serialization/deserialization of the checkpointed state from the set-up phases. Figure 4.5a shows that the serialization/deserialization to disk remains constant with respect to the VM size and accounts for a total of 110 milliseconds at worst. Fast C/R optimally scales with a multi-VM checkpoint/restore. Figure 4.5b shows 64 VMs (with 4 GB RAM each) checkpointed and restored in 0.21 seconds (for the slowest VM), a duration close to the fast C/R time of a single 4 GB VM.

Our current proof-of-concept saves and restores from the disk the second-level entries of the page-tables mapping VM memory (see Section 4.4.3). Figure 4.6 confirms the negligible impact that this operation has: around 15 milliseconds for a 256 GB instance.

Memory Preserving Reboot Analysis Leveraging warm-reboots saves an important fraction of time with respect to traditional machine reboots. We define the host OS switch time as the time between the invocation of the



(a) Single VM checkpoint/restore time.



(b) Multi VM checkpoint/restore time.

Figure 4.5: Hy-FiX fast C/R time breakdown.

Machine	**Reboot (s)**	`kexec` **(s)**
768GB, 2xIntel E2-2680 (2.40GHz)	143.27	21.88
256GB, 2xIntel E5-2630 (2.20GHz)	99.86	15.23

Table 4.4: Machine reboot vs. `kexec`

`reboot` system call until the initialization of the user-space (when QEMU starts). We compare the host OS switch time between a traditional machine reboot and a warm-reboot (`kexec`). The test is performed on two different host configurations: the host reported in Table 4.3 and a smaller 256 GB dual Intel Xeon E5-2630. The results reported in Table 4.4 show that the warm-reboot reduces the OS switch time by $\tilde{6}$.5x compared to the traditional approach.

Figure 4.7 shows the different warm-reboot duration on the same host as in Table 4.3, with a progressively larger RAM. We conclude that memory initialization has the largest impact on the host OS switch time. The host configured with 768 GB of RAM takes 12 seconds more than the 64 GB configuration, almost twice the time to reboot.

Hy-FiX lazy host memory initialization reduces the warm-reboot duration and makes it nearly immune to the growth of host memory. Figure 4.8 depicts the OS switch time with the lazy procedure enabled, evaluated for different

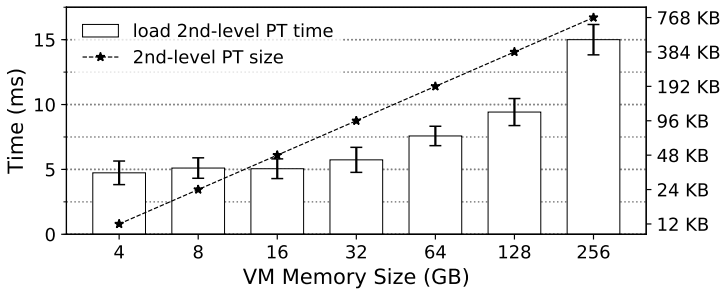


Figure 4.6: Second-level page-table load time from disk.

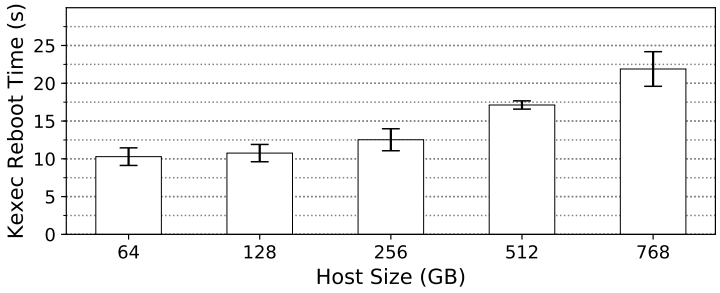


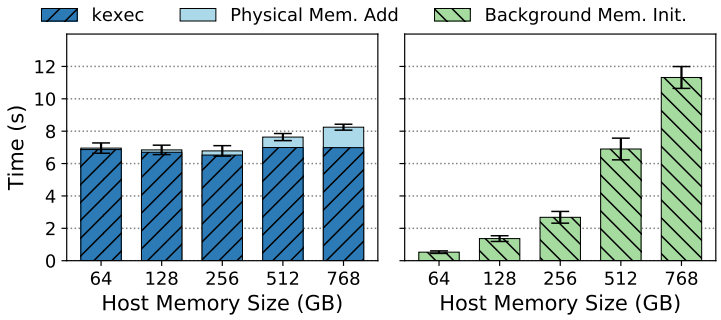
Figure 4.7: OS switch time w/ `kexec`.

host RAM sizes. The time remains constant throughout, differently from what we measure in a traditional `kexec` reboot in Figure 4.7. Indeed, with lazy memory initialization the host OS initializes only the 16 GB of safe memory region at boot (Figure 4.8a), leaving the rest to be processed on-demand and in the background (Figure 4.8b). The only remaining size-dependent operation is the *physical memory addition*. The kernel registers the memory within sysfs and initializes an identity mapping for the added physical range. These operations are however lightweight (Figure 4.8a).

4.5.2 Impact on Memory Access Latency

Hy-FiX immediately resumes the VMs as soon as the host OS reboots, before the host memory is fully initialized. The on-demand memory initialization reduces the reboot time at the expense of a temporary increase in the average memory access latency. We analyze this overhead by measuring the performance of the in-memory key-value store Redis [16].

Using the Yahoo Cloud System Benchmark (YCSB), [45] we issue a series of read-requests, each asking for a random (Zipf-distributed) 2 KB record stored within one of the 16 deployed Redis instances, hosted on as many VMs. Each virtual machine is provisioned with 32 GB of RAM, loaded with key-value records for an aggregated 512 GB Redis deployment. The VMs are



(a) OS switch time. (b) Memory initialization time.

Figure 4.8: OS switch time w/ lazy host memory initialization.

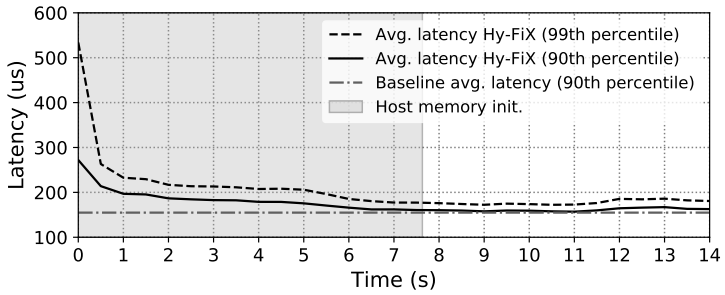


Figure 4.9: Redis average slow read-request after a Hy-FiX upgrade.

hosted on the 768 GB machine presented in Table 4.3. Halfway through the benchmark, we issue the upgrade for the hypervisor. We record the real-time evolution of the response latency after the VMs are resumed. To analyze the worst case of Hy-FiX, we decided to look at the 90% or the 99% of requests experiencing the highest response latency.

Figures 4.9 shows the results averaged over 0.5-second intervals. The sudden restart of the virtual machines causes a spike in the request latency which flattens within the next 500 milliseconds. The trend steadily decreases from being 40% higher, at 0.5-1 second, to 13% higher, at 4.5-5 seconds, eventually hitting baseline levels upon completion of the host memory initialization (after 7.6 seconds for 512 GB). Note, the 99th percentile follows the same trend. Another evidence that shows the performances quickly resuming after the restoration is visible in Figure 4.10. We grouped the 90th percentile latencies in 3 temporal windows. The first window within the first three seconds shows that 1% of the requests take twice the time to complete compared to the baseline. Starting from 4 seconds, the latency distribution is indistinguishable from the baseline. As more memory is lazily initialized, fewer and fewer memory accesses trigger the on-demand initialization procedure. Hence, the

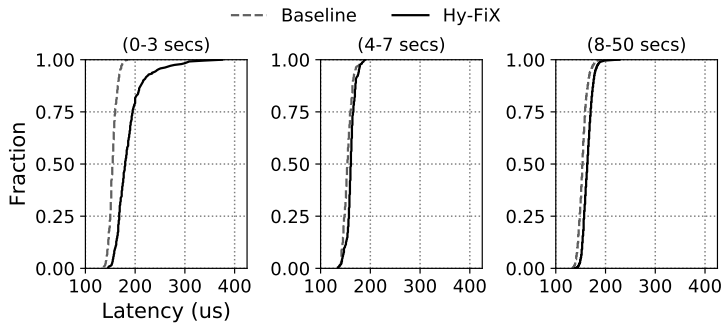


Figure 4.10: Latency distribution of Redis slow read-request (90th percentile), grouped in three intervals since the upgrade.

average access latency quickly converges to the original value.

4.5.3 Hy-FiX Memory Overhead

Hy-FiX demands memory to operate. Such additional memory is allocated for (*i*) the safe memory region, and the (*ii*) preserved memory bitmap.

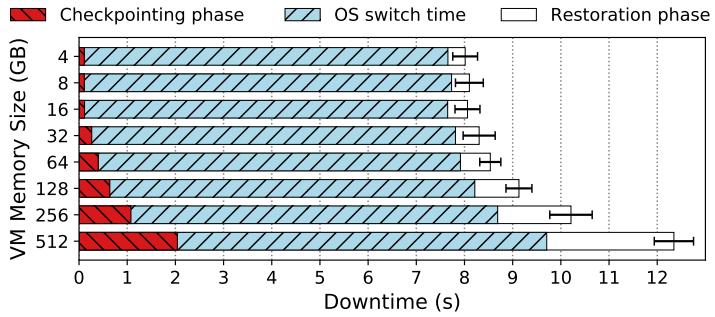
The safe memory region contains the new host OS during its booting phase. Its recommended size mainly depends on the memory installed in the host. In our experiments, 1 GB is the minimum size to store a Linux image, including an additional 32 bytes for every frame installed in the system. The rationale is to have enough space to allocate the `mem_map`, a large kernel structure we introduced in Section 4.4.4 that, indeed, holds a 32-byte descriptor for every frame. The persistent memory bitmap requires one bit to map each frame in the host. For a 1 TB host, the bitmap size is 2 MB. Note, the bitmap can be stored contiguously in a 2 MB huge-page.

For a 1 TB host, in total, 9 GB must be reserved for the safe memory region and 2 MB for the bitmap. The memory overhead of Hy-FiX represents 0.88% of the total host memory.

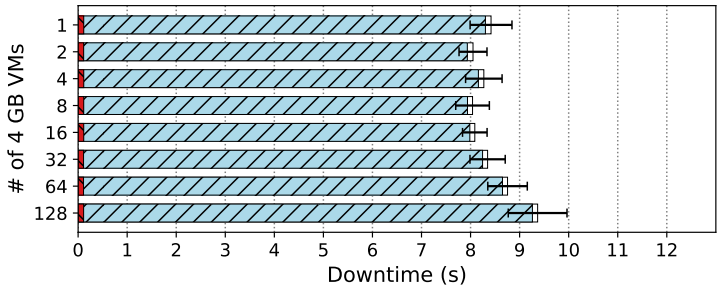
4.5.4 Hy-FiX Upgrade Time & Downtime Analysis

We define the Hy-FiX upgrade-time as the elapsed time from the start of the checkpointing for the first VM until the restoration of the last VM. Figure 4.11a reports an upgrade-time that ranges from 8.14 seconds to 12.34 seconds, for a single 4 GB and 512 GB VM respectively. In the multi-instance scenario (Figure 4.11b), the upgrade-time is less than 9.37 seconds for 128 VMs of 4 GB each (total of 512 GB of guest memory).

The host OS switch dominates the total upgrade-time. The host OS switch accounts for up to 62% of the total upgrade time (with a 512 GB VM), and 95% (with 4 GB instance/s). The host OS switch time is independent of the



(a) Single VM running.



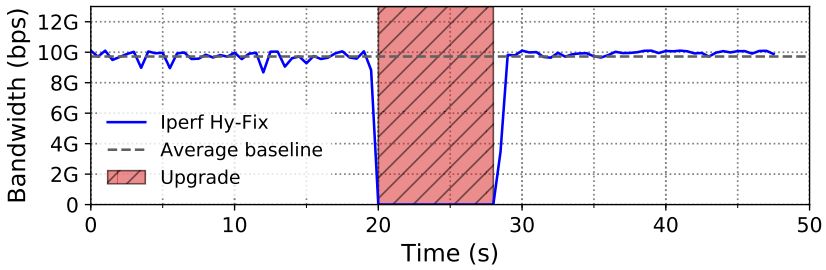
(b) Multiple 4 GB VMs running.

Figure 4.11: Hy-FiX upgrade time breakdown.

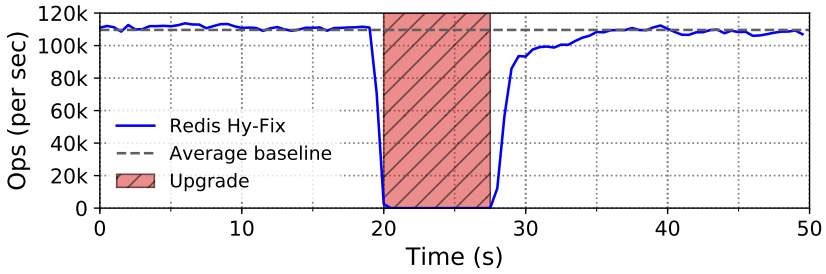
host memory usage and the number of running guests. As a consequence, Hy-FiX is weakly affected by the host load and consolidation, in contrast to alternative approaches that copy/transfer the memory (i.e., live migration). If we compare Hy-FiX upgrade time to the reload time of a 120 GB in-memory database, as reported in [52], our proposition would resume the service within 13 seconds, against the 2.5-3 hours expected for VM termination, restart, and database re-population.

The virtual machine downtime coincides with Hy-FiX upgrade-time. We show this by measuring the real-time throughput of two network applications, Iperf and Redis, during an upgrade. Iperf is a tool to performs TCP bulk transfers between a client and a server. We deploy Iperf inside a 32 GB VM running on our 768 GB host. The setup used for Redis is the same as the one presented in Section 4.5.2. The clients for both applications are within the same data center on a different host. The results are shown in Figure 4.12. The throughput for Iperf and Redis suddenly drops to zero upon checkpointing the VMs. About 8 seconds later the VMs are restored and the throughput suddenly ramps to pre-upgrade levels. Notice that Redis is initially affected by the host memory initialization as discussed in Section 4.5.2.

We must mention that during the experiment of Figure 4.12, the clients stop transmitting new requests once the blackout is detected. They restart



(a) Iperf throughput.



(b) Redis throughput.

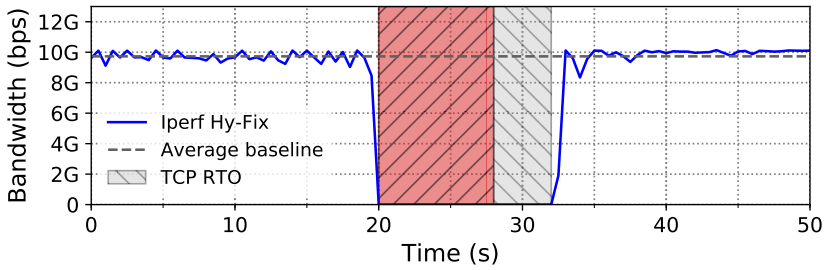
Figure 4.12: Throughput during a Hy-FiX upgrade.

only when VMs are detected available. For completeness, we also report the experiment in which the clients keep on retransmitting during the blackout. When the network packets of a request time out, according to TCP Retransmission Timeout (RTO), the new retransmission is sent after twice the RTO (exponential backoff). In such a specific case the perceived downtime by the clients is around 12 seconds as shown in Figure 4.13. It is worth stressing that the virtual machines are truly available once the upgrade completes. However, clients may experience a different downtime based on factors such as application-level timeouts, network protocols, and the sending time of the request with respect to the on-going hypervisor upgrade.

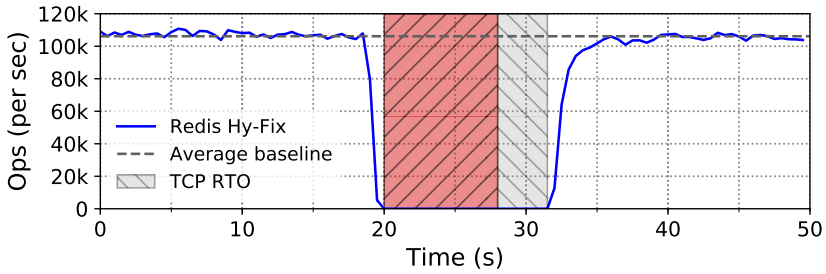
4.6 Discussion

In this section, we discuss the limitations of Hy-FiX and the possible directions that can be explored to improve the solution.

Hy-FiX enables the complete replacement of the software deployed, notably the host OS and the VMM. However, there are requirements to comply to enable the preservation of the virtual machines' state across the upgrade. (*i*) The new version of the VMM must be compatible with the virtual hardware state (forward compatibility). (*ii*) The representation of the VM memory must not



(a) Iperf throughput.



(b) Redis throughput.

Figure 4.13: Throughput during a Hy-FiX upgrade w/ clients retransmitting.

change. In Section 4.3, we made the assumption that the VMM always comply with (*i*), as popular instances do (e.g., in QEMU-KVM). Regarding (*ii*), we stated in Section 4.3 that VM memory representation is independent of the software version. However, there are exceptions: compressed memory and encrypted memory (e.g., Intel Software Guard Extensions—SGX [8]). Currently, Hy-FiX can efficiently restore the virtual machines if their memory is preserved *as-is*. Supporting scenarios where the memory representation changes requires plug-ins that convert between the formats, involving operations that may be lightweight (e.g., setting an encryption key), or a time-consuming serialization/deserialization of the data (e.g., adopting a new compression format).

Upgrade-wise, Hy-FiX leverages warm-reboots, which skip the hardware reset and firmware phase of the machine reboot. Therefore, maintenance tasks that require powering off a node, e.g., replacement of a hardware component, cannot be performed. Certain firmware upgrades are possible across warm-reboots, most notably flashing the CPU microcode. This class of upgrades turned out to be critical in recent years to mitigate the infamous Spectre and Meltdown vulnerabilities. CPU microcode updates are possible at runtime but are recommended to be applied at an early stage of the OS boot (at the earliest time possible) to maximize coverage [7], making Hy-FiX particularly suitable for the task.

Hy-FiX leverages the emulator routines to checkpoint/restore the VMs' hardware state (see Section 4.3.1). Therefore, Hy-FiX cannot operate on VMs with devices unsupported by the VMM. For instance, QEMU does not support the checkpoint/restore of directly assigned devices. As discussed in Section 2.3.1, this limitation is typically overcome by leveraging the hot-plugging capabilities of the guest OS and QEMU. The problematic devices are removed before the checkpoint and re-attached after the restoration [100].

Ultimately, the 7.6 seconds of Hy-FiX OS switch time dominate the downtime, representing a good candidate for further optimization. With lazy memory initialization, the strategy of deferring the execution of the time-consuming memory initialization dramatically shortened the reboot. One might wonder if other sub-tasks of the OS reboot could be treated the same way. However, it gets clear by inspecting the boot log (e.g., `dmesg` in Linux) that there are hundreds of diverse tasks that contribute evenly to the whole reboot delay. Initialization of interrupts, buses, sensors, PCI/PCIe devices, NICs, SCSI controller, as well as subcomponents such as the scheduler, the file-system, the networking, etc., are simply too many to be tackled independently. In [32], the authors devised a mechanism to boot multiple bare-metal instances of Linux within the same host, leveraging the software partitioning of the hardware (CPUs, memory, and devices) to create independent sub-domains where multiple OSes can run. This proposition is the object of our follow-up work on the same subject. In Chapter 5, we present this approach of booting the upgraded hypervisor in parallel to the running one, leveraging the same mechanism designed for Hy-FiX to transfer the VM state to the new instance without incurring any OS switch time.

4.7 Summary

We presented Hy-FiX, an in-place upgrade mechanism to apply generic fixes at the host OS and/or user-space-level of a KVM-based hypervisor such as QEMU-KVM. Hy-FiX easily integrates within the existing KVM hypervisor architecture. It requires only limited modifications to the Linux kernel (mostly in two kernel modules) and leverages the entire checkpoint/restore path commonly implemented by existing emulators.

The two key components of Hy-FiX, namely, the memory preserving reboot (with the lazy memory initialization), and the fast C/R, avoid the hardware resets typical of standard reboots, enabling the quick booting of an upgraded host OS. VMs are promptly recovered by simply mapping their virtual RAM to the memory content preserved across the reboot, hence avoiding costly copies from the disk or the network. As a result, Hy-FiX upgrades a 768 GB enterprise-class host in less than 13 seconds, no matter the workload, the memory in use, and the total number of running instances. By working in-

place, Hy-FiX requires no external resource or spare capacity in the DC, making it able to simultaneously upgrade an arbitrarily large number of hosts.

Chapter 5

Co-located Hypervisors for Efficient Live Upgrades

Contents

5.1 Introduction

At the core of any hypervisor there is a *kernel*. For type-2 hypervisors the distinction is clear: the VMM runs as a set of services hosted on a host OS which kernel provides the CPU scheduling, memory, I/O, and process

management (Section 2.1.1). For type-1 hypervisors the separation is blurred as VMM services are possibly implemented as kernel routines. Regardless of this distinction, an upgrade that involves replacing a hypervisor's kernel requires rebooting the physical host.

In Section 2.5.2 and Section 4.2, we presented the approaches to minimize the disruption caused to virtual machines because of the upgrade-induced reboot: *migrations* and *in-place upgrades*. Migrations evacuate the VMs before rebooting the host. This approach consumes the network bandwidth to transfer the VMs' state (especially the RAM), while also locking additional computational resources on the destination hosts to accommodate the displaced VMs. In-place upgrades operate within the host boundaries. As they require no pre-booking of resources, in-place upgrades lead to large-scale upgrades (e.g., fleet-wide zero-day vulnerability fixes), and, because VMs are never relocated, no external resource is needed. However, VMs experience a longer downtime in comparison to live migration due to the inability of the hypervisor to keep virtual machines running whilst rebooting. State of the art in-place upgrades for hypervisors leverage warm-reboots (i.e., `kexec` in Linux [59]) to minimize the reboot/recovery time [87, 66, 85]. The advantage of warm-reboots is twofold. First, the reboot duration is significantly shorter because both the *hardware reset* stage and *firmware execution* stage (BIOS/UEFI) are skipped [49]. Second, VMs can quickly resume within the warm-rebooted hypervisor by recovering their bulky RAM content directly from the host RAM.

Despite the advantages of warm-reboots, the total virtual machine downtime is still in the order of seconds. Virtual machines get stopped before the hypervisor begins to shut down. Several tasks are then performed to prepare the VMs for the post-reboot recovery (e.g., partial checkpointing of VMs' state [87]). Subsequently, the hypervisor progressively terminates all the subsystems (scheduling, I/O, etc.), eventually switching to executing the new hypervisor via a `jmp` instruction (hence, warm-reboot). The new hypervisor takes over and initializes all its subsystems, preparing the environment to run virtual machines. Only at this stage the VMs can resume. This whole procedure involves many tasks (depending on the hypervisor, the cloud platform, etc.), that last an arbitrarily long time. The VMs remain unavailable for up to 20 seconds, even on optimized production-grade solutions [85].

Keeping the old hypervisor running while a new hypervisor boots—in the same physical host—might considerably reduce the long downtime due to the host warm-rebooting. With such a solution, virtual machines can live migrate from the old hypervisor to the newer one only when it is fully initialized and ready to host them. Hence, VMs would experience the same downtime as live migration, whilst remaining within the same host (in-place upgrade). Furthermore, if memory is shared between the two hypervisors, the migration

procedure does not need to transfer the RAM content of the guests, avoiding the inconvenience of live memory transfers or remote memory (see pre-copy and post-copy live migration in Section 2.3.1).

Unfortunately, two hypervisors cannot simply run side-to-side on the same physical hosts. Only a single bare-metal system can run on a typical computer platform (e.g., x86-64, ARM), as demonstrated by the existence of server virtualization itself, which, at its core, allows multiple legacy OSes to execute on top of a hypervisor over the same underlying hardware. *Nested Virtualization* enables running two or more hypervisors within the same host, at the expense of two major drawbacks [35]. First, VMs run by a nested hypervisor incur in a 10%-20% decrease in the performance (higher CPU usage [48], lower I/O throughput [5]). Second, the bare-metal hypervisor (commonly referred to as *L0*) cannot be upgraded without rebooting the host.

In this Chapter, we introduce *Multi-FiX*, a solution to fully upgrade a hypervisor, leveraging a second bare-metal hypervisor that executes next to the primary one, on the same physical host. Multi-FiX fully boots the upgraded hypervisor while the current one temporarily remains alive and capable of running VMs, keeping them available. The two hypervisors share the portions of the host memory where the RAM content of the guests is mapped, enabling an efficient zero-copy migration with minimal downtime. Our main contributions are:

- Demonstration of how, inspired by the *multi-kernel OS* design [34], and the principle of *partitioning the hardware resources* [32], a generic hypervisor can be modified in order to support a *multi-kernel boot*. That is to say, a hypervisor becomes capable of booting and running next to another existing hypervisor. Multi-FiX design leverages mainly the same technologies behind Hy-FiX, our previously presented warm-reboot-based solution, namely software-booting a new system, and hot-plugging/hot-unplugging CPU, RAM and I/O devices. We implemented the modifications to support a multi-kernel boot on the QEMU-KVM hypervisor for x86-64.

- Design of an in-place upgrade schema for hypervisors to operate arbitrary upgrades (kernel and—optionally—VMM, if type-2 hypervisors), taking advantage of the two bare-metal hypervisors to protect the virtual machine *network availability* during the upgrade. We implemented a proof-of-concept for the QEMU-KVM hypervisor for x86-64.

- Evaluation of the network downtime, and the computational performance, of VMs during the upgrade. We show that VMs experience minimal downtime in the order of tens-of-milliseconds.

The remainder of the chapter is structured as follows. Section 5.2 presents the solutions that are mostly related to Multi-FiX. Section 5.3 presents the

technical background. Section 5.4 discusses the design of the multi-kernel boot, while Section 5.5 presents the in-place upgrade schema. Section 5.6 details the implementation of Multi-FiX. Section 5.7 evaluates the solution. Section 5.8 presents limitations, and Section 5.9 concludes the chapter.

5.2 Related Work

We refer to Section 2.5.2 and Section 4.2 for a detailed classification of the different approaches to upgrade a hypervisor. Here we focus on the closest solutions to Multi-FiX, namely upgrades based on nested virtualization. We also cover the basics of multi-kernel OSes, a fundamental design for Multi-FiX. Finally, we discuss the relationship between Multi-FiX and Hy-FiX, our in-place upgrade solution based on warm-reboots.

5.2.1 Nested Virtualization

Nested virtualization enables two or more full-fledged hypervisors to run within the same physical host. A single hypervisor runs on the bare-metal (L0), with one or more hypervisors (each referred to as L1 hypervisor) hosted inside VMs managed by L0. In turn, the L1 hypervisors run their own set of VMs, hence the name nested virtualization. With nested virtualization, the upgrade strategy consists of moving the VMs from an L1 hypervisor to another newly instantiated L1 hypervisor, co-located on the same host. Solutions like HyperFresh [48] and VMBeam [67], leverage the aforementioned schema to perform upgrades with minimal downtime in the order of tens of milliseconds, no matter the RAM size and workload of the VMs. Thanks to the L1 hypervisors sharing the same portion of physical RAM, guest memory does not get serialized and transferred during the migration, decoupling the instance kind (large or small size, active or idle workloads) from the downtime. This special feature is implemented via memory page remapping, the same technique adopted in Hy-FiX to quickly recover virtual machine memory after the warm-reboot. As the two L1 hypervisors are configured with access to the same memory where the guest RAM is stored, it is sufficient to share the virtual-to-physical mapping to migrate memory between the two hypervisors. This is simply achieved by exchanging the root address of the page-tables that describes the mappings. The two hypervisors do that via a virtual communication channel implemented by the underlying L0 hypervisor. We refer to this migration technique as *zero-copy migration*.

Nested virtualization suffers from two major shortcomings when used for hypervisor upgrades. First, it adds further overhead in terms of higher I/O latency and CPU utilization due to the second virtualization layer. [35] and [48] propose several optimizations for L0 to close the performance gap between

nested and non-nested virtualization. Despite the research efforts, production nested virtualization still incurs a 10% performance penalty for CPU-bound workloads [5]. [48] reports a 20% increase in the CPU utilization over the non-nested baseline to achieve line rate in TCP bulk-transfers, an I/O workload known to be lightweight in terms of cost of packet processing [83]. These overheads permanently affect the workloads. Lastly, the complexity of all the optimizations to reduce the overhead at L0 is non-trivial and adds up to the hypervisor's codebase. Hence, L0 upgrades may be unavoidable, therefore requiring a physical host reboot.

Multi-FiX adopts the same strategy of having two co-located hypervisors, sharing memory to perform zero-copy migrations between the two systems. However, Multi-FiX does not leverage nested virtualization to run the multiple bare-metal hypervisors on the same host. Hence, (*i*) Multi-FiX avoids the performance degradation outside the upgrade-time; (*ii*) Multi-FiX upgrades can patch every software component deployed inside the hypervisor, from kernel to VMM; (*iii*) despite the mandatory kernel modifications to support our multi-kernel boot, the rest of the upgrade technique leverages standard VMM (e.g., QEMU) capabilities, for instance, to perform the migrations.

5.2.2 Multi-kernel Operating Systems

Multi-kernel OS is the name of an operating system architecture originally introduced by *BarrellFish OS* [34]. A multi-kernel OS is formed by multiple independent bare-metal kernels running on each CPU of a multi-core platform, communicating only via a message-passing protocol. The main objective of BarrellFish is abstracting the design of OSes from any hardware-dictated synchronization mechanism, e.g., the cache-coherence protocol. BarrellFish allows the transparent replacement of each kernel, without incurring any disruption at the OS, or application, level [99]. The latter is also the main objective of Multi-FiX. However, BarrellFish is based on the overhauling of traditional OS designs. On the other hand, Multi-FiX aims for easy integration with existing type-1/2 hypervisors.

Several independent projects have demonstrated how a monolithic kernel, such as Linux, can be converted to become a multi-kernel OS, bringing the same benefits of BarellFish onto mainstream systems [32, 60, 75]. These projects are all based on the key technique of *partitioning the hardware resources*. The *non-shareable* hardware resources are sliced into partitions, each containing enough CPUs, memory, devices, etc., for the independent kernels to boot and run. The objective is to avoid implementing a complex coordination mechanism among kernels. Practically, the legacy kernels are modified to discover and initialize only the resources within their partition. The partitioning is purely software, each kernel instance voluntarily respects these imposed boundaries. In Multi-FiX, we strip down the multi-kernel OS architecture

for a precise use-case: *temporarily running two bare-metal hypervisors within the same host.* We provide the minimum requirements to implement, and enforce, the partitioning of the hardware resources that enable this scenario. In contrast with [32, 60, 75], Multi-FiX ultimately dismisses the multi-kernel environment at the end of the upgrade. Eventually, only the upgraded hypervisor remains in execution with full control over the complete host hardware.

5.2.3 In-place Hypervisor Upgrades and Warm Reboots

In-place upgrade solutions for hypervisors capable of replacing arbitrary kernel code require rebooting the host. Solutions like Hy-FiX, previously presented in Chapter 4, leverage warm-reboots to speed up both the reboot process and the preservation/recovery of the running VMs. In this Section, we provide a detailed insight on warm-reboots, the key technology that powers Hy-FiX. Furthermore, we revise the key architectural points behind Hy-FiX, as the same mechanisms enable the efficient migration of VMs between the co-located hypervisors of Multi-FiX.

Warm-Reboots. Assume that the binary image of a new system is contiguously loaded onto a reserved memory area. As depicted in Figure 5.1, warm-reboots comprise three stages:

(i) *System shutdown.* The current system is properly shut down. The first step is to terminate all the processes and tasks. Then, drivers get invoked to quiesce the relative devices (stopping DMA transactions, interrupts, unregistering device). Ultimately, all the CPUs are halted, except for a single one (i.e., CPU#0 on x86-64) where the next stages take place.

(ii) *System Switch.* This stage depends on whether the objective is to power off, reboot, or warm reboot the machine. For warm-reboots, the thread of execution on the last CPU, after properly setting the CPU registers for the transition, performs a `jmp` instruction, setting the program counter to executing at the new system's entry point in its binary image. The old system ceases to exist, replaced by a new one, which starts as if booted by a traditional boot-loader.

(iii) *System initialization.* The booting system must acquire knowledge of the hardware resources available and usable. The firmware and boot-loader provide a partial description of the whole platform. This information is either encoded in RAM/ROM stored tables (e.g., ACPI tables [39]), or directly forwarded to the new system in accordance with a particular boot protocol (e.g., Linux boot-time parameters [18]). Some hardware is dynamically discovered by probing specific platform registers. It is the

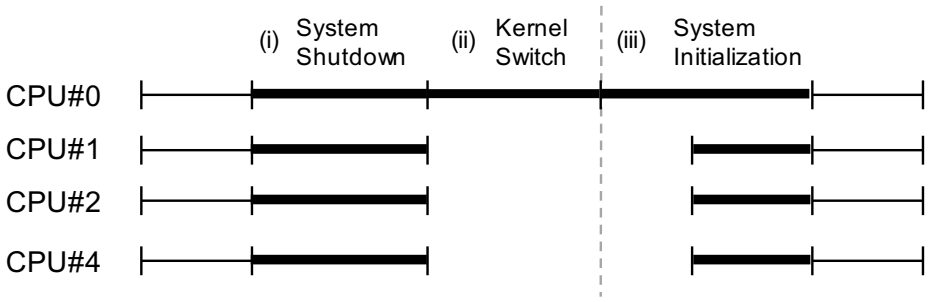


Figure 5.1: Execution threads during a warm-reboot.

case, for instance, for PCIe devices. Once the hardware is discovered and enumerated, the new system initializes it and makes use of it.

Hy-FiX. Three are the main components of Hy-FiX. The first, *fast checkpoint/restore* (fast C/R), is responsible for preserving the execution state of the running virtual machines across the warm reboot. The virtual hardware state of the guests (vCPUs, vNICs, etc.) gets serialized into a stream of bytes and dumped to non-volatile memory. As compared to the legacy virtual machines checkpoint/restore, the fast C/R is quicker because the bulky guest virtual RAM is not dumped to storage. Instead, such RAM content remains resident in the host memory, surviving unaltered across the warm reboot. This is thanks to the *memory preserving reboot*, a protocol for memory management agreed between the old and the new booting system to protect the host RAM ranges where the data to preserve is located. Finally, *zero-copy memory re-linking* takes the raw host RAM preserved during the warm reboot and reconnects it to the resumed virtual machines. The latter technique leverages the recovery of the page-tables, preserved as well during the warm-reboot, revealing all the mappings between virtual machine pages and host frames.

Multi-FiX performs the same class of upgrades as Hy-FiX (host kernel and VMM), operating in-place within a single host. It leverages the core tools of Hy-FiX to quickly relocate the VMs between the two hypervisors. However, Multi-FiX achieves a downtime three orders of magnitude shorter than Hy-FiX.

5.3 Technical Background: Modern x86-64 Computer Platforms

The description for Multi-FiX design targets the Intel x86-64 architecture. In this Section, we introduce the characteristics of modern hardware commonly found in enterprise-grade data center hosts. Features of such hardware are key for the design of Multi-FiX.

CPUs and xAPIC. Multi-core computer platforms are standard in data center hosts. Each CPU is equipped with circuitry that allows any I/O device to notify events via interrupt signals to any arbitrary CPU. The same circuitry provides each CPU with a ticking timer and enables the CPUs to send Inter-Processor Interrupts (IPIs) to other CPUs. This chip is called *Local Advanced Programmable Interrupt Controller* (LAPIC). This is part of the *xAPIC*, the latest architecture for programmable interrupt controllers. Aside from the LAPICs embedded in the CPUs, a computer platform features one, or more, IOAPIC. The IOAPIC is specialized in routing *legacy interrupts* to the right destination CPU. By legacy interrupts we refer to any interrupt technology that is not *Message Signalled Interrupts* (MSI), the signaling mechanism used in PCIe. Typically, IOAPIC incurs more latency in delivering interrupts. Hence, slower interfaces leverage it for signaling the occurrence of events.

PCI Express (PCIe). PCIe is the de-facto standard fabric to connect high-throughput low-latency I/O devices to the CPU-memory subsystem. PCIe devices are identified by a hierarchical ID system known as *Bus-Device-Function* (BDF). PCIe devices are dynamically discovered by reading/writing an ensemble of memory-mapped I/O registers called *PCIe configuration space*. The software accesses the PCIe configuration space to probe each combination of the BDF addresses to discover the actual presence of a peripheral behind. PCIe devices support Direct Memory Access (DMA), enabling any device to independently read/write from/to the RAM. Furthermore, PCIe leverages MSI (or the more advanced MSI-X), an in-band mechanism to send interrupt signals over PCIe messages. MSI enables each PCIe device to independently send interrupts to the LAPIC of any CPU in the computer, bypassing the IOAPIC. PCIe messages, whether carrying DMA operations (memory read and writes) or an MSI message, have as a target a host physical address. Such an address can be transparently remapped via the IOMMU.

IO Memory Management Unit (IOMMU). The I/O Memory Management Unit is responsible for remapping the target addresses of read/write PCIe transactions (DMA operations and MSI interrupts). As presented in Section 2.1.4, the IOMMU is the fundamental hardware component that enables a safe direct device assignment to the VMs.

Each PCIe device can have a different memory mapping. The mappings are contained in RAM-stored page-tables, indexed by the BDF address of the PCIe device. The software selects the mapping by writing to an IOMMU register the pointer to the page-table root, analogous to the `CR3` register for the MMU.

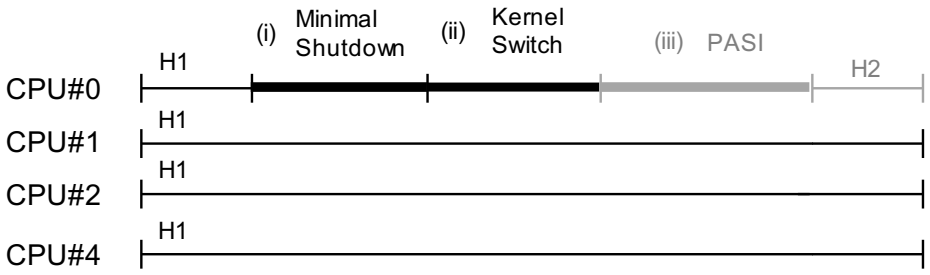


Figure 5.2: Execution threads during a multi-kernel boot.

5.4 Multi-kernel Boot

Warm-reboots constitute a solid ground to build our novel *multi-kernel boot*. In Multi-FiX, the objective is to boot a new system (i.e., the upgraded hypervisor), while the current one (i.e., the outdated hypervisor) remains capable of performing useful work (i.e., running VMs). We define this procedure as multi-kernel boot. Warm-reboots partially offer the same service: a new system is booted out of the currently running one. However, the old system is also shut down during the process.

There are two challenges to tackle in order to successfully transform a warm-reboot into a multi-kernel boot. First, the system that boots shall have enough computational resources—*CPU and memory*—to execute the boot procedure and fully initialize. Second, the two co-located systems must avoid concurrent accesses to *non-shareable hardware resources* (e.g., I/O devices) which may disrupt the execution on both systems. We achieve a multi-kernel boot by replacing the stages (*i*) and (*iii*) of a traditional warm-reboot (as presented in Section 5.2.3), with, respectively, a *minimal system shutdown* that frees the hardware resources for the new system to execute the boot, and a *Partition Aware System Initialization* (PASI) that limits the hardware resources acquired during the system initialization. PASI is based on a predetermined *partitioning of hardware resources* to make sure that each system knows which resource can be safely accessed after the multi-kernel boot takes place. In addition, we introduce the technique of *hardware resource migration*, to gradually dismiss the old system by transferring the control of all the hardware resources to the newly booted system.

In the remainder of the chapter, we refer to the system running before the multi-kernel boot as *H1* (as in *Hypervisor#1*), and the second system booted afterwards as *H2* (as in *Hypervisor#2*). Figure 5.2 provides a high-level overview of the multi-kernel boot procedure from the CPU perspective.

5.4.1 Partitioning of Hardware Resources

In order to co-locate two independent bare-metal systems on the same physical host, we slice the hardware resources into two partitions. Each system exclusively accesses the resources within its partition. This design principle avoids the implementation of complex synchronization mechanisms to multiplex the non-shareable hardware resources among two uncooperative systems. In the scope of Multi-FiX, this allows legacy hypervisors to be easily converted in supporting the multi-kernel boot.

Each system is instructed on the boundaries (e.g., which CPU a system is allowed to run on) and voluntarily respects these limitations by avoiding the discovery and initialization of any resource outside its partition. The partitioning schema we devised targets four categories of hardware resources: (*i*) *CPUs*, (*ii*) *memory*, (*iii*) *I/O devices*, (*iv*) and the *chipset*. We now detail each category in relationship with the ultimate goal of running two co-located hypervisors.

CPU Partitioning. *We partition the host CPUs assigning* CPU#0 *to H2 and the remaining CPUs to H1.* On Intel x86-64, OSes start executing on CPU#0, the only active CPU at boot. In a multi-kernel boot, H2 starts via the kernel switch procedure that eventually performs the `jmp` instruction into the new system. Thus, the execution continues on the same CPU where the jump took place. In order to simplify the implementation of multi-kernel booting to a legacy hypervisor, we perform the kernel switch on CPU#0. As such, H2 boots on CPU#0 as during a legacy boot. A single CPU is also enough for H2 to execute the whole boot.

Memory Partitioning. We subdivide the memory in three categories: *private RAM*, *shared RAM*, and *shared MMIO*.

H1 and H2 are both assigned with non-overlapping ranges of physical memory where their RAM is mapped and sized properly to allow each system to boot and run. We call this memory private RAM. Although RAM is a *shareable resource*, un-coordinated reads/writes result in corrupted data for both systems. Each system is assigned with a private RAM area, large enough to accommodate the whole system binary image, and to provide enough memory that the system can allocate for its internal usages (i.e., not to run VMs as is shared). Note that H1 is assigned with a private RAM area. This enables subsequent upgrades with the roles of H1 and H2 swapped.

H1 and H2 are further assigned with the same ranges of physical memory that maps the RAM meant to be shared. We call this memory shared RAM. In the context of hypervisor upgrades, the guest memory is shared to enable the efficient zero-copy migrations of VMs. We identify a set of RAM ranges, disjoint from the private RAM described before, designated for the memory

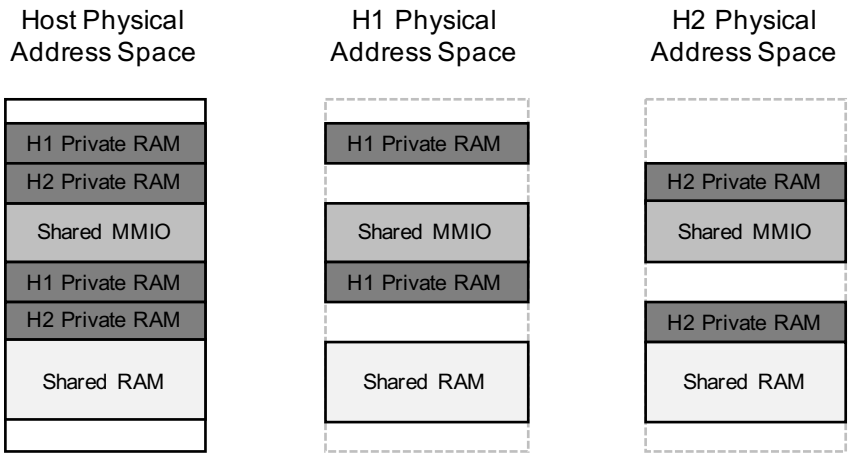


Figure 5.3: Partitioning schema for the memory.

sharing between the co-located systems. Disjoining the private RAM from the shared RAM ensures that H2 does not inadvertently corrupt the content of the shared memory during the early stages of its initialization. This implies that the system memory management during the boot needs no patching, provided that the shared RAM ranges can be hidden from H2.

H1 and H2 are both assigned with every physical address range mapping any hardware resource. We call this memory shared MMIO. The host physical address space includes memory-mapped hardware resources such as firmware tables (e.g., ACPI tables) and platform registers (e.g., PCIe configuration space, IOAPIC registers, IOMMU registers). Some registers, e.g., firmware tables, are read-only, hence easily shareable. Although both systems do have access to these memory-mapped registers, the exclusive assignment of the remaining hardware resources (I/O devices and chipset) guarantees that no collision can take place.

Figure 5.3 portrays the partitioning of the memory as described above.

Partitioning of I/O Devices. *Legacy devices (management low-throughput high-latency) are assigned to H2, whereas core devices (high-throughput low-latency) remain assigned to H1.* Typically, high-throughput low-latency I/O devices are *PCIe devices* leveraging DMA and MSI interrupts. Conversely, low-throughput high-latency devices can be connected via different bus technologies to the CPU-memory subsystem, e.g., keyboard via USB. Non-PCIe devices, and PCIe devices that use legacy interrupts, do not support the features that allow a device to be migrated, at run-time, between the two co-located systems. In the scope of hypervisors, we assume that all the devices relevant to the I/O workload of the virtual machines (either in PCI passthrough, or back-ends of the emulation) are exclusively *PCIe devices lever-*

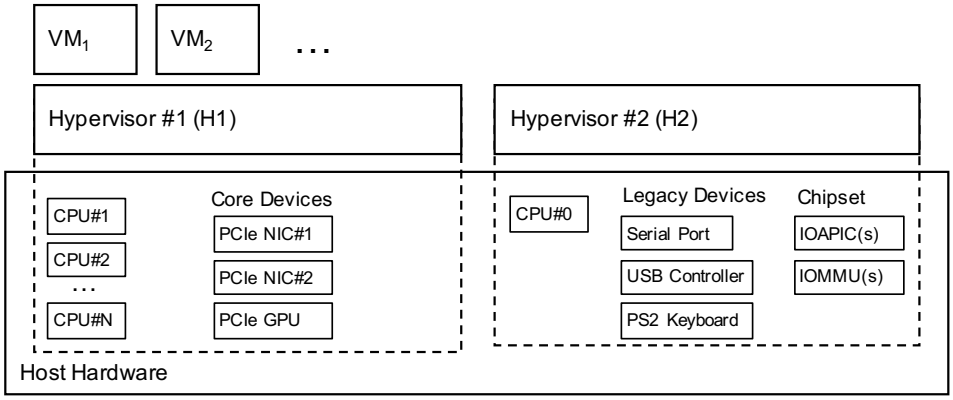


Figure 5.4: Partitioning schema for the hardware resources.

aging MSI. We call this class of I/O peripheral *core devices*. We also assume that the remaining devices, non-PCIe, and PCIe with legacy interrupts, are solely used for host management, e.g., providing access via the serial console and the keyboard. We call this class of I/O peripheral *legacy devices*. When H2 boots, H1 retains access, and control, of all the core devices, allowing VMs to keep performing their I/O operations.

Chipset. *H2 is assigned with the control and management of the IOAPIC(s) and the IOMMU(s).* IOAPIC and IOMMU are both responsible for re-routing interrupts (legacy and MSI respectively). The IOMMU also remaps the addresses of DMA operations. Both chips are configurable via software by writing the content of the routing/remapping tables. Because of our previous assumption, core devices only leverage MSI and do not leverage the IOAPIC to route their signals. Hence, H2 can safely acquire full control of all the IOAPICs in the platform. This partitioning schema adopts the unmodified initialization of the IOAPIC as it happens during a legacy boot. On the other hand, core devices require a functioning IOMMU to keep interacting with H1, performing DMA operations towards H1's buffers and notifying events via MSI. *The IOMMU mappings are preserved intact by H2 for all core devices still in use by H1.*

Figure 5.4, and Figure 5.3 recap the partitioning of CPU, memory, I/O devices, and the chipset between H1 and H2.

5.4.2 Minimal System Shutdown

The minimal system shutdown takes place on H1 before performing the kernel switch that sets H2 in execution. The objective of this procedure is to release the hardware resources, currently held by H1, destined to H2. Compared to a traditional shutdown, this minimal one avoids the termination of any running

process or task. The resources released from H1 are CPU#0, the legacy devices, and both IOAPCI and IOMMU. H2 private RAM is already reserved and off-limits within H1.

Releasing CPU#0 requires *offlining* the CPU. This operation involves stopping the scheduling of processes/tasks on a given CPU, followed by the migration of all the interrupt handlers registered in that CPU in favor of other online ones. Once CPU#0 is offlined, H2 can be started without affecting H1, which considers such CPU as unavailable. Typically, modern OSes (e.g., Linux) readily provide APIs to online/offline arbitrary CPUs, used in the context of CPU hot-plugging/hot-unplugging.

Legacy devices are released by quiescing them via their drivers' shutdown method, followed by the deregistration of the device, i.e., retire the APIs to access it. The device stops any ongoing DMA operation and interrupt signal, then, it sets the hardware registers to a device-specific configuration that allows a new driver to re-initialize the device. This procedure is the same that takes place during the legacy shutdown phase of a warm-reboot. Indeed, the drivers within the newly booted system are in charge of re-initializing the devices and make them usable.

Releasing IOAPIC and IOMMU does not require a driver shutdown to clear their state. H2 simply performs the reset on its own. However, to prevent H1 from further altering the configuration on both chips, we declare a new system state called `multi-kernel-booted`. A system in `multi-kernel-booted` cannot perform the following operations: (*i*) *re-scanning busses* (e.g., PCIe bus) or discovering new I/O devices, and (*ii*) *re-balancing interrupts*, i.e., migrating interrupt handling to different CPUs. These operations all involve the modification of the routing/remapping tables of the IOAPIC and/or IOMMU by H2.

5.4.3 Partition Aware System Initialization

During its initialization, a legacy system performs the *discovery*, and *initialization*, of all the hardware resources available in the computer platform. We must prevent H2 from re-initializing and utilizing resources outside its designated partition. *Note that a system does not initialize resources that are not discovered.* This is the principle to design the partition aware system initialization.

On Intel x86-64, the resource discovery varies from platform to platform. Usually, the firmware encodes a description of the available CPUs, NUMA layout, and physical address layout, via ACPI. Aside from parsing this information, the booting system performs some probing on its own, such as exploring the PCI bus, and other plug-and-play peripherals such as SATA/SAS busses. In Multi-FiX, H2 acquires control of most of the hardware available to the platform, leaving H1 with few homogeneous resources: the CPUs, H1

private RAM, the PCIe core devices. The choice of immediately granting H2 the ownership of most of the devices reduces the number of subsystems in H2 to patch to prevent the unwanted acquisition of resources.

In the case of Linux (QEMU-KVM hypervisor), the system provides a convenient interface, namely the zero-page structure [18], that allows the boot-loader to pass certain information to the booting system. This includes passing the layout of the physical address space, useful to hide to H2 the presence of H1 private RAM, and, at least initially, the shared RAM regions. Furthermore, via the command-line parameters many hardware resources can be excluded and marked unusable. For instance, the number of CPUs to initialize can be limited via the option "`ncpus`". By leveraging the aforementioned interface, we can influence the hardware discovery operated by H2 without the need of patching any kernel code. However, probing PCIe devices does require a small patch to forbid the discovery and initialization of the core devices still in use by H1.

5.4.4 Migration of Hardware Resources

The partitioning of resources between H1 and H2 is temporary. Eventually, H1 terminates, and all its hardware transitions to H2 that ultimately becomes the single system running on the host. We now describe the procedures to migrate the ownership of CPUs and PCIe core devices from one system to the other. These are the last resources controlled by H1 after H2 boots. Note that the private RAM is never handed over to the other system.

CPU Migration. CPU migration leverages the same CPU offlining procedure used on CPU#0 during the minimal system shutdown (Section 5.4.2), followed by the symmetrical *CPU onlining* operation that sets another H2 kernel thread in execution on that CPU. We now describe both the procedures on x86-64.

Offlining a CPU consists of disabling scheduling and interrupt handling on the target CPU, eventually confining it to an infinite loop that executes the instruction `mwait` (to save energy). Symmetrically, CPU onlining consists of H1 sending a wake-up interrupt sequence (INIT-SIPI-SIPI) which frees the offlined CPU from its sleep loop. The awakened CPU becomes fully online after starting the thread of execution (that H2 specified) which schedules the processes of H2. H1 and H2 synchronize during the completion of offlining/onlining via a simple low-throughput high-latency communication channel based on shared memory. Note, CPU onlining/offlining is readily available on mainstream legacy OSes.

Core Device Migration. By our assumption, core devices are high-end PCIe devices, natively supporting PCIe hot-plugging. This core device migration

simply consists of hot-unplugging the device within one system and hot-plugging it on the other. H1 and H2 synchronize via their communication channel. However, this migration procedure takes time. The resource remains unusable for the whole quiescing-time and re-initialization. We measured the resulted unavailability between 1.5 seconds and 4.8 seconds for the tested NICs (see Section 5.3). CPU migration incurs downtime as well, although lasting only milliseconds.

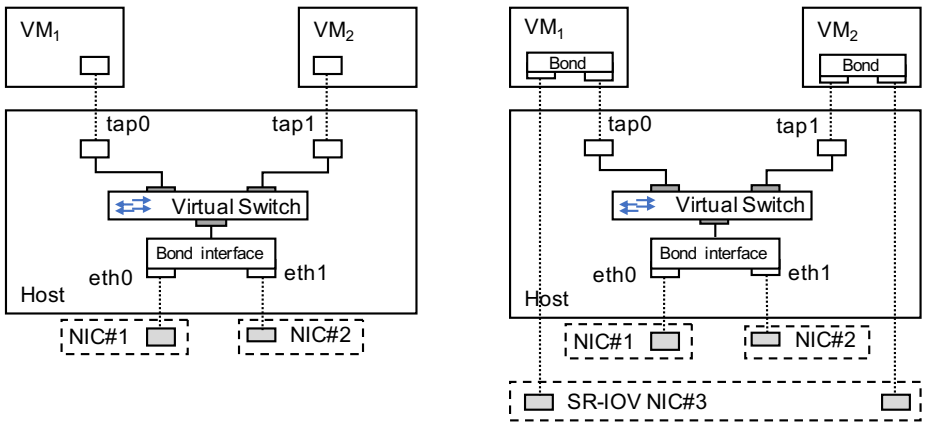
5.5 In-place Upgrade Strategy

The multi-kernel boot enables to fully boot an upgraded hypervisor, referred to as H2, running side-by-side with the old hypervisor, referred to as H1. While H2 fully initializes, H1 keeps running the VMs with no performance degradation except for CPU#0 that works for H2. In order to finalize the upgrade, two tasks remain to be performed: (*i*) virtual machines must relocate from H1 to H2, and (*ii*) H2 must take control over all the host hardware resources, becoming the only bare-metal system running. However, migrating a PCIe device makes it unavailable for several seconds, jeopardizing the execution of the VMs whose I/O workload leveraged it. We propose a coordinated migration of VMs and hardware resources specialized for hosts with redundant Network Interface Controllers (NICs) to preserve the *virtual machine network availability* during the upgrade. We name this procedure *migration stage*.

We define VM network availability as the Ethernet/IP connectivity between a VM and the production network. Under the assumption that the hypervisor leverages the network to provide the VMs' storage (e.g., via NFS, iSCSI, FCoE, NVMe-oF, etc.) the network availability also covers the storage availability. In this Section, we describe the migration stage, and then host and hypervisor configurations supported.

5.5.1 Virtual Environment and Hardware Redundancy

In Multi-FiX, the migration of a PCIe NIC between the two co-located hypervisors causes the device to become unavailable. Therefore, we require the host and the hypervisor to provide *NIC failure tolerance*. Multi-port NICs, i.e., NICs with multiple individual Ethernet ports, only provide link-failure tolerance. In order to achieve the stronger NIC-failure tolerance, the host must feature at least two distinct PCIe NICs, both connected to the production network. Figure 5.5a illustrates the host network configuration with two NICs (both assumed to be single-port for simplicity). The Ethernet links of each NIC are aggregated into forming a single logical *bonded interface* (e.g., via Linux bonding drivers). The bonded interface must be configured in *load-balancing* or *active-backup* mode to provide the transparent failover in case of



(a) Paravirtualized interfaces. (b) Passthrough and paravirtualized NIC.

Figure 5.5: Network configurations supported by Multi-FiX.

NIC failure, diverting all the traffic through the healthy interface [12].

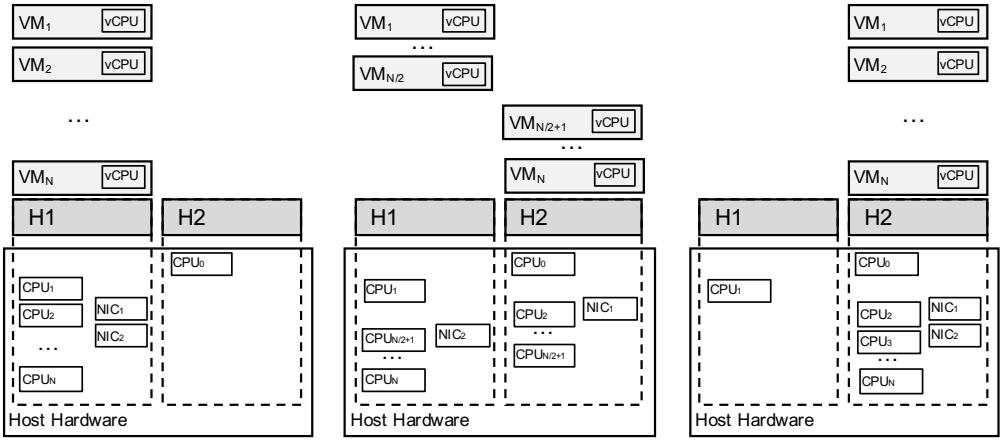
PCI Passthrough is supported via a paravirtualized NIC, as explained in Section 2.3. The passthrough NICs are detached from the VMs during the upgrade via hot-unplugging. The traffic is then redirected through the paravirtualized NIC via a bonding interface in the guest. When the upgrade is done, the passthrough NIC is re-attached. Figure 5.5b shows a passthrough scenario supported by Multi-FiX. Note that once passthrough NICs are detached, the host network configuration is identical to the scenario presented in Figure 5.5a.

5.5.2 Migration Stage

Assume H2 has booted via the multi-kernel boot. Figure 5.6a shows a snapshot of the situation at this stage. All the VMs are running on H1, and the hypervisors respectively control their slice of hardware resources according to the partitioning schema described in Section 5.4.1. At this point, we enter the Migration Stage. The VMs and hardware resources are migrated in a precise order to protect the network availability of the VMs. As PCIe NICs remain unusable for several seconds while re-initializing, *their migrations are offset with respect to each other.*

First, a single NIC migrates from H1 to H2. Virtual machines remain reachable via the second NIC. When the relocated NIC fully initializes within H2, a subset of VMs is immediately migrated to balance the network load between the two hypervisors. As VMs move, a subset of the CPUs migrates along, to power the computational needs of the displaced guests. This concludes the first phase (*phase 1*) of the migration stage. Figure 5.6b depicts a snapshot of the host at this point.

The second phase (*phase 2*) starts with the migrations of the remaining



(a) Before phase 1. (b) After phase 1. (c) After phase 2.

Figure 5.6: Snapshots of the host during the migration stage.

VMs to H1. Similar to phase 1, the remaining CPUs on H1 (except one) migrate along with the VMs. When all the migrations are complete, the last NIC on H1 is relocated to H2. H2 now controls all the host hardware and runs all the VMs. Figure 5.6c shows the final snapshot at the end of phase 2. The upgrade ends with H1 offlining its last CPU, hence terminating its execution.

The exact number of VMs and CPUs that migrate during phase 1 depends on the resource demand of the running guests. The set of VMs and CPUs to migrate is determined as follows:

- Assume each of the two NICs is assigned to a single hypervisor.
- The set of VMs migrated together contains guests such that *the aggregated bandwidth demand of all instances in the set is greater or equal to the bandwidth offered by a single NIC.*
- The set of CPUs migrated together contains a number n of CPUs such that *the aggregated CPU demand of all instances in the aformentioned VM set is satisfied, at its best, by the computational power of n CPUs.*

These rules guarantee load balancing at each phase of the migration stage, to avoid major performance penalties during the upgrade.

5.5.3 Hy-FiX Integration

The shared memory between the two hypervisors enables an efficient live migration of virtual machines between the two co-located hypervisors. In presenting Hy-FiX, we introduced two techniques to enable the fast restoration of VMs after a hypervisor warm reboot: the *fast VM checkpoint/restore* and the *zero-copy memory re-linking*. The VMs are stopped and checkpointed by

What	Version	# Files	LoC Modified
Linux	5.4.91	21	844
Hy-FiX Linux Modules	5.4.91	9	1330
`kexec-tools`	2.0.22	7	114

Table 5.1: Multi-FiX code analysis.

serializing their virtual hardware state into a standardized representation, allowing the rewriting of its format to adapt to newer hypervisor versions. The guest's RAM is not checkpointed. Upon restoration, VMs are linked back to their RAM by remapping their virtual address space to the physical addresses in the host where the RAM content is. Multi-FiX borrows the fast VM checkpoint/restore and the zero-copy memory re-linking to implement the *zero-copy live migration*.

Zero-copy live migration is a standard *post-copy* live migration modified to avoid the transferring of the guest memory (as in Hy-FiX fast VM checkpoint/restore). Because of shared memory, guests' RAM is already available on H2. Upon initialization, H2 loads the page-tables that describe the virtual mapping for each VM, making the memory accessible for the future migrated VMs. Zero-copy live migration incurs the same minimal downtime of post-copy live migration (in the order of milliseconds), but without the overhead to fetch the remote memory.

5.6 Implementation

We implemented a proof-of-concept of Multi-FiX for QEMU-KVM on Intel x86-64. The hypervisor is based on Linux 5.4.91 and QEMU version 5.0.0. Table 5.1 reports the lines of code modified in the legacy components, i.e., the Linux kernel, and `kexec-tools`, including also the added kernel modules inherited from Hy-FiX. The whole solution, including the zero-copy live migration, works with unmodified QEMU.

Partitioning of Hardware Resources. We implemented the partitioning of the hardware resources via a set of scripts that explores the available hardware on the host and computes the subdivision that future multi-kernel boots will leverage. These scripts are run on H1 before the multi-kernel boot takes place. For core devices, the output file is a list of BDF addresses indicating which PCIe devices stay with H1 after the multi-kernel boot. This list is passed to H2 to serve as a *blacklist* of PCIe devices not to discover.

Memory partitioning takes place before any hypervisor is ever booted. The private RAM, the shared RAM, and the shared MMIO ranges are identified.

When H1 first boots, the private RAM of H2 is already reserved. The private RAM of each hypervisor always contains 512 MB below the 4 GB mark, to enable 32-bit devices to perform DMA operations. Additional private RAM is provisioned according to the same rule we established for Hy-FiX and its safe memory region (see Section 4.5.3). As a reference, for a 128 GB host, we reserved 2 GB for each hypervisor. In addition, we reserved 64 KB to store a bitmap that indicates which pages within the shared RAM are currently in use. This bitmap is part of the protocol inherited from Hy-FiX to share the memory between the two systems.

Minimal Shutdown. We implemented the minimal shutdown leveraging Linux CPU hot-unplugging and PCIe hot-unplugging. Since Linux for x86-64 version 3.8, CPU#0 can be offlined as all the other CPUs. The hot-unplugging is performed by writing the value zero to "`/sys/devices/.../cpuX/online`". Legacy Linux takes care of the whole offline procedure applying the steps we presented in Section 5.4.4.

Concerning devices, in our PoC we only shut down the non-core-device PCIe devices. We perform this operations by writing a non-zero value to "`/sys/bus/pci/devices/.../BDF/remove`", where `BDF` is the address of the PCIe devices to shut down.

Kernel Switch. We introduced the term "OS switch" in the context of warm-reboots (Section 5.2.3). We rename such an operation in "kernel switch" to emphasize the transition of a single kernel threads from executing instructions of H1 to executing instructions of H2. The kernel switch is based on Linux `kexec` and `kexec-tools`. In Multi-FiX we modified `kexec` to perform the following: (*i*) load the binary images directly at the target address in H2 private RAM, and (*ii*) start executing H2 without performing the legacy system shutdown. Our patched `kexec`, after the loading stage, sends a non-maskable interrupt to the offlined CPU#0, trapped in the `mwait` loop. The legacy Linux offlining for CPU#0 places a specific interrupt handler to break the `mwait` loop when the CPU needs to be onlined again. We modified such a handler to execute the `kexec` kernel switch routine in case of a multi-kernel boot. Similar to what we did for Hy-FiX (Section 4.4.1), we also patched `kexec-tools` to customize the Linux *zero-page*, to pass information to the booting system.

Partition Aware System Initialization. The only kernel subsystems we patched to implement the partition aware system initialization are (*i*) *PCIe probing*, and (*ii*) *Intel IOMMU initialization*.

We leverage the Linux *boot-time command-line*, set by `kexec`, to both inform the booting system that it is starting as a special multi-kernel boot,

and to constraint its execution on CPU#0 via the option "`ncpus=1`". We leverage the *zero-page* to force a synthetic view of the available memory. In particular, we achieve this by altering the `E820` table, used by the boot-loader to inform the booting kernel on the physical address space layout (e.g., ranges mapping RAM, ranges reserved area, ranges not mapped). We mark H1 private RAM as *reserved*, and we omit the ranges of shared RAM, hiding and protecting them while H2 boots.

We patched Linux probing of PCIe devices to skip the discovery and initialization for specific BDF addresses. We simply introduced a check to avoid probing the PCIe configuration space registers corresponding to BDF addresses that are blacklisted.

Regarding the preservation of the IOMMU state, Linux initialization for Intel's IOMMUs readily provides a special initialization path for a `kdump` boot (debug kernel booted via `kexec` upon panic [54]). This special path preserves the current IOMMU configuration to protect the new kernel from rogue DMA transactions fired by unaware devices trying to communicate with the crashed kernel. In our PoC, we enabled this code path during a multi-kernel boot.

Migrations of Hardware Resources. CPU migration also leverages Linux CPU hot-plugging. The implementation is identical to what describes for the minimal system shutdown. The target CPU is offlined via the interface "`/sys/devices/.../cpuX/online`". Once the routine completes, we signal H2 to online the same CPU via the same interface "`/sys/devices/.../cpuX/online`".

Core device migration is also implemented leveraging Linux PCIe hot-plugging, in the same way as described in the minimal shutdown to free the legacy PCIe devices. We leverage "`/sys/bus/pci/devices/.../<BDF>/remove`" to logically un-plug a device. Once the routine completes, we signal H2 to complete its part of the migration. The BDF address of the device is removed from the blacklist of PCIe device not to discover, and then the bus is re-scanned via the API "`/sys/bus/pci/rescan`". As the un-plugged devices are re-discovered, the drivers load and make the peripheral available.

Memory Sharing & Zero-Copy Migration. We integrated the Linux modules that manage the reboot persistent memory in Hy-FiX. Note that in Multi-FiX we seek for a similar across-reboot persistence of the RAM, both for protecting such memory during the booting of H2 and later for recovering the mappings to make the memory accessible.

The module within H1 records the allocated memory pages in shared RAM, and notes for each VM the root addresses of the page-table describing its mapping to physical memory. The list of addresses is saved onto H2's booting-

Server	Dell PowerEdge R630
CPU	2 x Intel Xeon E5-2630 v3 (2.40GHz, 8 cores)
RAM	128 GB
NIC#1	Intel 82599ES 10Gbps (dual-port, driver: `ixgbe`)
NIC#2	Intel X520 10Gpbs (dual-port, driver: `ixgbe`)

Table 5.2: Testbed node specifications.

ramdisk. Once H2 fully initializes, a module loads the page-tables from their root addresses and recreates the virtual address spaces for each VM. For each recovered virtual address space, the module exposes it as a memory-mappable (`mmap`) virtual device file.

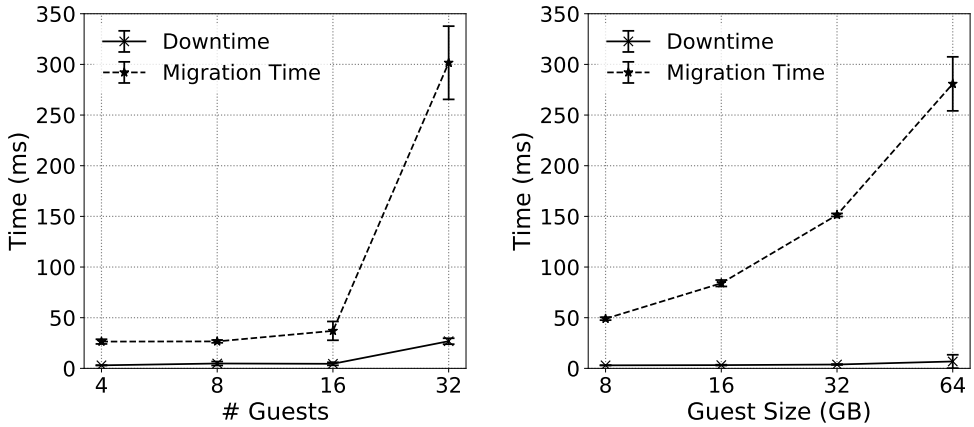
We leverage an option of QEMU to back, with a file descriptor, a VM's memory, using the virtual device file exposed by our kernel module. The virtual machines are re-created with full access to their memory, paused until the rest of their state is sent via migration. The migration is a standard QEMU live migration over TCP. In our proof-of-concept, the network traffic passes between the NICs available to each hypervisor (at this stage the resources are partitioned as in Figure 5.6b). QEMU live migration is invoked with the option `x-ignored-shared` to avoid sending any memory marked as shared, i.e., the entire guest RAM.

5.7 Evaluation

We evaluated the effect of a Multi-FiX upgrade over the running VMs. All the experiments have been conducted on Grid5000 [30], over machines with the hardware specs reported in Table 5.2. The host network is configured as in Figure 5.5a, with paravirtualized interfaces in the guests, connected to an Open vSwitch virtual switch, and an Open vSwitch bonding interface configured in *Source Load Balancing* mode (SLB bonding), aggregating two interfaces, one from each of the two NICs. The cluster network comprises a single 10Gbps switch, connecting all the machines involved in the experiments (a server machine and a client machine).

5.7.1 Zero-copy Migration Downtime Analisys

In our PoC, zero-copy migration is based on legacy QEMU live migration, with the main difference of how guest memory is transferred. As the hypervisors share all the guest memory, zero-copy migration does not transfer any byte of guest memory. The zero-copy migration downtime is computed on H1 by QEMU as the time elapsed between stopping the VM (to transfer their virtual hardware state), and the sending of the end-of-file signaling the ending of



(a) As function of guest number.

(b) As function of guest RAM size.

Figure 5.7: Guest downtime for zero-copy migration.

migration.

Zero-copy migration is almost immune to the increase in both the number of guests and the RAM size of each guest. Each guest is configured with 1 vCPU, 1 GB RAM (when the value is fixed), 1 vNIC, a virtual serial port, and a keyboard. We create a synthetic memory dirtying workload on the VMs to prove that zero-copy migration is immune to the dirty memory rate. We adopt Redis [16], an in-memory database server, and the Yahoo Cloud Service Benchmark (YCSB) [45] to generate a mixed read/write requests targeting zipfian distributed keys. Each request reads/writes an 8 KB record.

When multiple migrations take place simultaneously, the worst values are collected as representative of the trial. Figure 5.7a shows the downtime as function of the number of guests, and Figure 5.7b as a function of guest RAM size. Our zero-copy migration incurs the same downtime as post-copy live migration or pre-copy live migration when the WSS is zero (see Section 2.3.1). Indeed, such downtime is less than 30 milliseconds, in line with the median downtime for 1 vCPU guests reported in [4].

5.7.2 NIC Reinitialization Time Analisys

The downtime PCIe NICs incur during the device migration encompasses the time to quiesce and re-initialize the device, plus the time to configure the OS network stack accordingly, making the interface usable. We benchmarked the downtime for the two NICs reported in Table 5.2, the same peripherals that our test-node embeds.

We estimate the total time by measuring the time elapsed between the start of the PCIe hot-unplugging to shut down the device, and the reception of a first ping echo reply. A client constantly probes the NIC, waiting for

NIC	Reinit. time (s)
Intel 82599ES (NIC#1)	3.5±0.05
Intel X520 (NIC#2)	1.79±0.05

Table 5.3: Average downtime time for PCIe NICs (Measurement tolerance 100 ms)

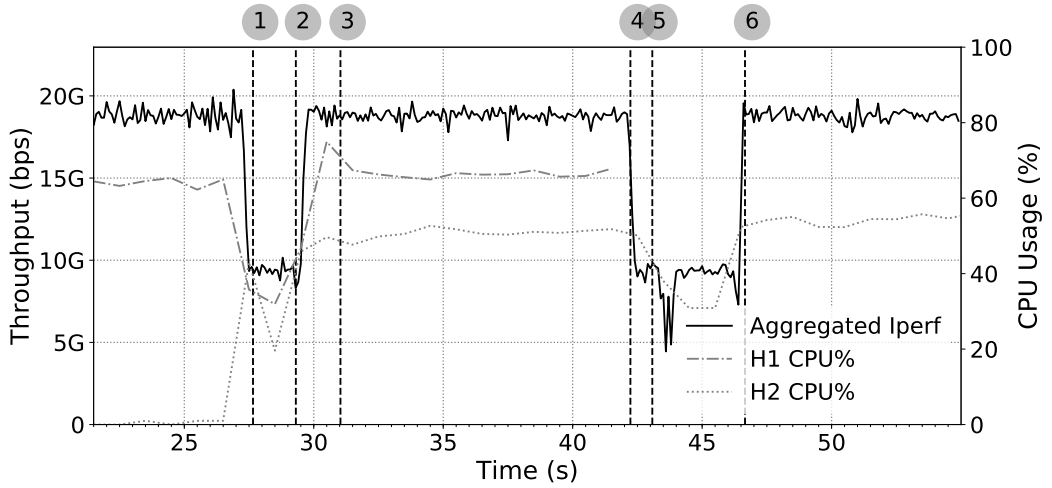
100 milliseconds between each ping echo request. Due to the rate at which the probes are sent, the experiment overestimates by at most 100 milliseconds the re-initialization time. Note that the RTT within the cluster is in the order of tens of microseconds. The results are reported in Table 5.3. The downtime for both NICs is in the order of seconds, with a noticeable difference between the two cards.

5.7.3 Impact on Guest Workloads

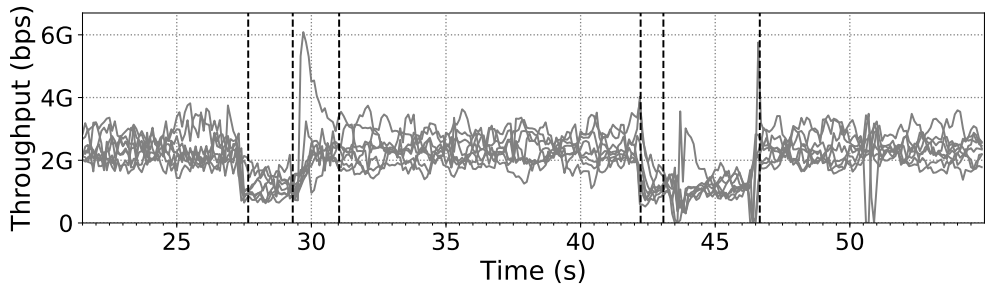
Multi-FiX preserves the network availability of the virtual machines during the upgrade. However, due to the relocation of the (*i*) NICs, (*ii*) CPUs, and (*iii*) VMs, the performance of the guest workloads is affected. In particular, relocating a NIC makes the device unavailable for 1.8-3.6 seconds, temporarily reducing the total host bandwidth by the card rate. Furthermore, as CPUs incur a brief downtime during their offline/online cycle, the total computational capacity can fluctuate during the migration stage. We study two workloads, a pure network workload—a TCP bulk transfer—and the mixed CPU-memory-network workload of an in-memory database server.

TCP Bulk Transfer. We analyzed the impact of Multi-FiX over long-lived TCP connections by deploying 8 VMs in a single node, each VM executing one `iperf` server instance. The host total bandwidth is 20 Gbps, provided by the two 10 Gbps NICs aggregated in a source-load-balanced bonding interface that distributes the traffic on a *per-flow* basis.

Figure 5.8a shows the *aggregated* `iperf` throughput during the migration stage of the upgrade (Section 5.5.2). The first phase of the migration stage starts by relocating the first NIC to H2. We see at ① a decrease of throughput by 50% due to the single remaining 10 Gbps NIC. Around 2 seconds after, the relocated NIC is again functional within H2, and half of the VMs join it resuming the total 20 Gbps of aggregated throughput at ②. Half of the CPUs start migrating alongside the VMs and finish at ③. We see an increase in the CPU utilization on H1 as the CPUs are offlined to join the VMs on H2. The second phase is symmetrical to the first one. It starts with the remaining VMs on H1 migrating to H2 The throughput drops again by 50% due to all the 8 VMs sharing a single 10 Gbps NIC at ④. The remaining CPUs (except one) also leave H1. Once the VMs are fully migrated at ⑤, the NIC on H1



(a) Aggregated



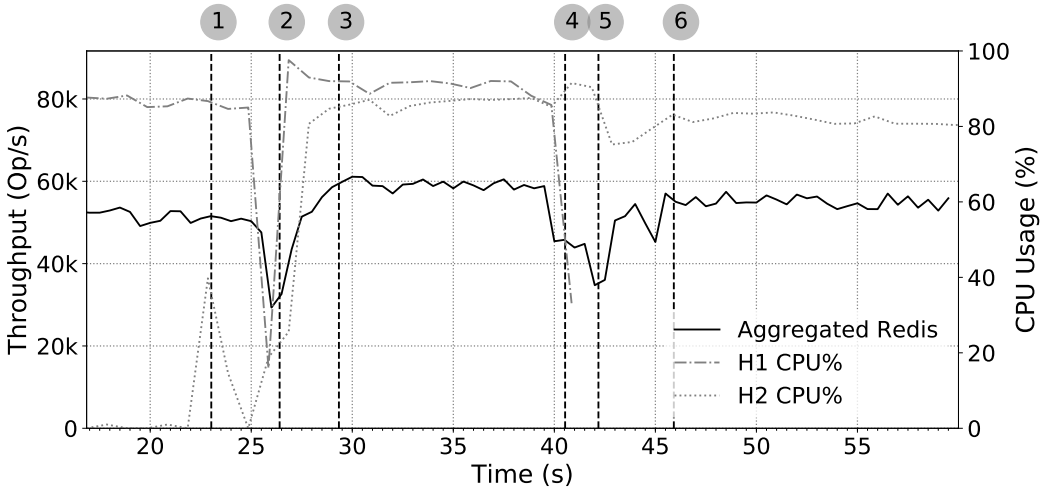
(b) Individual.

Figure 5.8: Iperf throughput during a Multi-FiX upgrade.

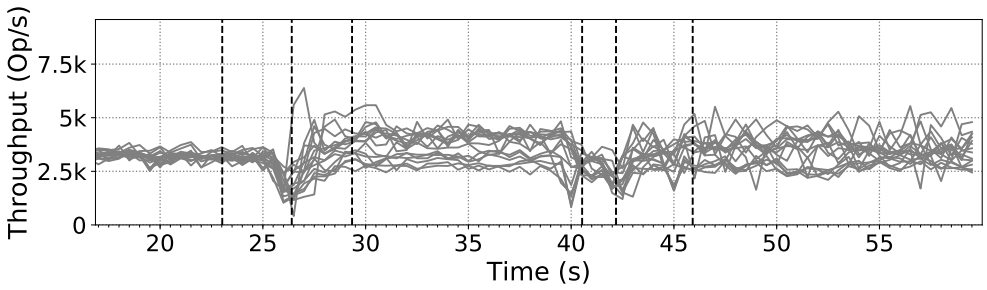
relocates to H2, taking around 3.5 seconds to reinitialize at ⑥. Eventually, the throughput resumes to 20 Gbps when both NICs and all VMs are on H2. Overall, the throughput of the TCP bulk transfer is halved for a total time of around 5 seconds, corresponding to the sum of the downtimes of each NIC. Note that, as depicted in Figure 5.8b, no VMs remains unreachable for an extended time higher than 100 ms.

In-memory Database. We adopt the aforementioned Redis [16] and YCSB [45] to generate the in-memory database workload. The experiment involves 16 VMs, each with 1 vCPU and 2 GB of RAM, hosting an independent Redis instance loaded with 64 thousand 1 KB records. The clients generate via YCSB mixed read/write requests targeting Zipf-distributed keys. Note that each request is blocking.

Figure 5.9a shows Redis's aggregated throughput during the migration stage of the upgrade. We built the experiment to put high CPU pressure on the host. Indeed, H1 shows an average CPU utilization above 80% before any



(a) Aggregated.



(b) Individual.

Figure 5.9: Redis throughput during a Multi-FiX upgrade.

migration starts. In this experiment, the network bandwidth does not constitute a bottleneck for the in-memory database workload, hence the departure of a NIC does not affect the throughput. However, there is a certain response to the change in the number of online/offline CPUs for a hypervisor. During the first phase of the migration stage, the relocation of the NIC at ① does not lead to any throughput drop. However, as half of the CPUs (CPU#2-8) and half of the VMs migrate to H2 at ②, we see the CPU utilization on H1 fluctuating and briefly hitting close to 100%. This noise is responsible for the brief decrease of throughput, but stabilizes quickly after the CPUs and VMs complete the relocation to H2. Similar behavior happens during the second phase, a fluctuation of the CPU utilization on both H1 and H2 shows up at ④, the moment when the remaining VMs and CPUs start migrating. Overall, despite the high CPU requirement of the benchmark, Multi-FiX achieves a moderate and brief degradation of an in-memory database throughput. Note that, as depicted in Figure 5.9b, no individual workload ever reaches zero.

5.8 Discussion

In this section we discuss the limitations of Multi-FiX and the future research directions that may overcome the major problems.

Multi-FiX currently only protects the network availability of the virtual machines during the upgrade, provided that at least two distinct PCIe NICs are installed and connected to the production network. This limitation stems from the long time a device takes to relocate between the two hypervisors. PCIe NICs take several seconds with great variability between models. The same will occur for NVMe controllers which, according to [97], incur a 1.2 and 2.5-second downtime, depending on the card. As no I/O operation can be performed by the device while re-initializing, the workload must be diverted to another peripheral. For network cards, it is simple. A second PCIe NIC stays available while the first one is re-initialized, taking over all the traffic as if a link failure took place on the restarting card. In Multi-FiX, we exploited this property of the NICs to preserve the guest network availability. However, stateful devices, such as NVMe drives, do not easily support a backup spare that can take over the workload in case of failure. RAID controllers are built as a single PCIe device, hence, all the redundancy offered by the RAID schema is lost when the controller is unavailable. GPUs and HBAs may suffer the same problem.

The reason for the re-initialization of the device, in order to relocate it between the two hypervisors, is the following: *the device driver in the destination system must rebuild a valid computational state compatible with the state of the device.* To work around the problem, *the driver state shall be either migrated or not reset at all.* We discuss this possible future direction in Section 6.2.

5.9 Summary

In this Chapter we presented Multi-FiX, a novel addition to the family of in-place upgrade solutions for hypervisors. Multi-FiX leverages multi-kernel boot, our novel software boot procedure that spawns two bare-metal uncooperative systems on the same physical hardware, leveraging warm-reboot and the partitioning of the hardware to avoid concurrent accesses to non-shareable resources. We implemented Multi-FiX for QEMU-KVM for x86-64, running a recent Linux version which required little modifications to few key subsystems (`kexec`, PCIe probing, IOMMU initialization, and CPU#0 onlining) to support the multi-kernel boots, leveraging mainly existing OS features (warm reboots, PCI hot-plugging, CPU hot-plugging). Our PoC was tested on a modern host, equipped with the same hardware normally found in contemporary production data centers. Multi-FiX delivers clear improvements over

Hy-FiX, applying the same class of upgrades while preserving the network availability of the running VMs, incurring downtimes as high as 30 milliseconds at the costs of sacrificing half of the host network bandwidth for a short time (8 seconds on the tested hardware).

Chapter 6

Conclusion and Future Directions

Contents

6.1 Conclusion

Server virtualization and consolidation are fundamental to decrease the operational costs of data centers. Migration is used to re-configure the placement of virtual instances (dynamic re-consolidation), with the main objective of regrouping sparsely unused resources and eventually deploy additional instances or power off empty physical machines. Furthermore, maintenance events leverage migrations to avoid the disruption incurred by turning off servers, network, power, and cooling equipment. Migration is not, however, the panacea. For idle virtual machines, dynamic re-consolidation, combined with traditional over-booking mechanisms, either falls short in reclaiming enough resources (e.g. with ballooning or page deduplication) or might lead to unpredictable performance degradation (a known downside of the hypervisor swapping technique). For hypervisor upgrades, the benefits of limited downtime obtained with live migration collide with the poor scalability of such a strategy, as live migration is a resource-demanding process in terms of both network utilization and spare resource availability. Hence, hypervisor upgrades based on live migration potentially result in long upgrade campaigns that prolong exposure to fatal vulnerabilities.

In this thesis, we went beyond the classical migration techniques for virtual instances. We exploited the versatility of transplanting the execution state of virtual instances, in particular of virtual machines. With SEaMLESS, part of the VM's state is isolated and extracted to deceive end-users (or monitoring agents) into believing the VM is still up and running, while only the front-end part of the services is still active. The hypervisor can then reclaim the allocated (locked) resources by disabling the idle VM. With Hy-FiX and Multi-FiX, part of the VM's state is checkpointed, marshalled, and fed to an upgraded version of the hypervisor, whereas the remainder of the state,

the bulky guest RAM, is recovered from the environment (host RAM ; without any data transfer), efficiently preserving the VMs across the disruptive hypervisor restart. As the two key challenges identified at the onset of this thesis relate to idle virtual machines and hypervisor upgrades, that both consume high amounts of resources in the data centers, we summarize below our contributions to these two problems.

Discussion on Idle Virtual Machines in Data Centers. In this dissertation, we presented SEaMLESS, a solution to boost the consolidation in data centers populated with numerous idle virtual machines. Despite being idle, virtual machines retain their memory. RAM thus becomes the bootleneck when deploying more instances on a single physical machine. Simply terminating idle VMs to reclaim their memory undermines the availability (and violates, in a number of cases, SLAs) that data center operators are supposed to guarantee. Any strategy that tackles such a problem shall deal with sudden end-user requests that cease the idle virtual machines' inactivity, processing them within the shortest delay.

A variety of state-of-the-art solutions propose a mechanism to reclaim memory from running idle VMs. Via partial VM migration, the little portion of the idle VMs' state that makes their on-board services available, i.e., the working set, can be consolidated on a few physical machines, suspending the remainder of the hosts that hold the bulk of the state. The working set estimation for the idle execution, necessary to identify the portion of VM's state to maintain available, might however fail to account for keep-alive messages or other trivial requests. Therefore, receiving such a class of requests can unnecessarily prompt the restoration of the VMs. Proxies to replace the disabled idle VMs offer finer control over which end-user requests require resuming the offlined instances. However, the challenge is to implement a generic proxy that supports any network protocol (e.g., TCP, UDP, etc.), and application that run on idle VMs (web server, SSH server, etc.). For this reason, we proposed SEaMLESS, a framework able to migrate the gateway processes, constituting the entry-point to the VM's services, to a container. The gateway processes transplanted inside the container act as a proxy replacement for a disabled idle VM. The container hence represents a lightweight Virtual Network Function (VNF) that maintains the network presence of the services. User activity is detected at the container by monitoring the interaction between the transplanted gateway processes and their container, successfully identifying requests that require resuming the VM.

SEaMLESS enables the usage of various techniques to disable the VMs (whose gateway processes have been migrated) and reclaim their resources, especially the memory. Suspend-to-disk fully reclaims the entire memory but incurs a slow VM restoration, proportional to the size of the guest RAM.

Suspended-to-RAM VMs resume instantly but no memory is returned. We propose a combination of the best of both worlds, suspend-to-swap, that evicts most of the VM's to storage while restarting immediately. Suspend-to-swap leverages the hypervisor swapping, eagerly moving VM's memory to swap to make room for other instances, exploiting the lazy fetching of memory pages as they are needed.

Discussion on Hypervisor Upgrades. In this dissertation, we presented Hy-FiX and Multi-FiX to perform scalable in-place upgrades of hypervisors. The limited upgrade capabilities of the live upgrades and little scalability of migrations leave space for improvements in the domain of hypervisor upgrades. Restarting the hypervisor is the universal approach to upgrade the entire software stack deployed in a hypervisor, from the host OS (in type-2 hypervisors) to the VMM, including any additional user-space components. However, the disruptive hypervisor reboot makes the restarting approach a weak upgrade technique. Suspend-to-disk can preserve the VMs' state across the reboot, but incur a long incurs when saving/loading the bulky RAM of running the VMs.

The breakthrough is leveraging warm-reboots to shorten the hypervisor reboot time and enable an ideal forward-compatible zero-copy migration of VMs towards an upgraded hypervisor. Nevertheless, the warm-reboot time still dominates the virtual machine downtime. Our proposition, Hy-FiX, improved on state-of-the-art solutions by applying key modifications for legacy KVM-hypervisors to drastically reduce the long memory initialization that the host OS undergoes during the reboot. Hy-FiX's lazy memory initialization decouples the hypervisor restart time from the host memory size. The former becomes almost independent from the latter, a key feature as RAM size keeps increasing in large data center machines. Compared to live migrations, warm-reboots still incur a noticeable downtime three orders of magnitude higher, which led us to design Multi-FiX.

Booting a second hypervisor in the background while the primary one keeps executing is an approach explored in nested virtualization to perform efficient non-disruptive upgrades. However, nested virtualization penalizes the workloads due to the higher virtualization overhead. Inspired by the classical multi-kernel OS architecture, aiming at executing several weakly-cooperative OSes on the same hardware, we proposed a successor to Hy-FiX: Multi-FiX. The features of modern hardware, such as multi-core processors, xAPIC, PCIe devices, and IOMMUs, enable an easy implementation of a multi-kernel OS architecture for hypervisors, specialized to perform upgrades. By sharing some portions of the physical memory, the two hypervisors can perform a zero-copy migration of the virtual machines, handing the guest execution to the upgraded hypervisor. Hardware resources initially split among the two

hypervisors, migrate alongside the VMs. Eventually, the newer hypervisor remains in control of the physical machines, completing the upgrade. Multi-FiX achieves a complete in-place upgrade of a hypervisor while incurring a few tens of millisecond-long downtime, comparable to live migration.

In the next Section, we propose some future directions for SEaMLESS, Hy-FiX, and Multi-FiX.

6.2 Future Directions

Future Work for SEaMLESS. SEaMLESS is a refined way to over-book the host memory. As a consequence, a promising follow-up work shall study the integration of SEaMLESS with a platform manager like OpenStack [13]. SEaMLESS is able to promptly detect any new user-activity, hence data center operators can take the most suitable operation to resume the VM. In the presence of a memory hotspot, i.e., a flock of VMs resuming from idle at the same time, a VM cannot resume on its host due to the risk of memory contention and uncontrolled swapping. Live machine migration is helpful to resolve crowded situations, however, the choice of which virtual machine to migrate, and how, presents several challenges. For instance, pre-copy live migration does not immediately release the memory allocated by a VM, whereas post-copy progressively returns memory as pages are sent to the destination. The most appropriate choice depends on the available resources and the type of services hosted on the involved VMs. It calls for in-depth integration of the resource manager (Openstack or a platform-independent VM scheduler like BtrPlace [55]).

Future Work for Multi-FiX. The major shortcoming of Hy-FiX is the slow nature of the employed warm-reboot, which leads to downtime in the order of seconds. To compete with the unnoticeable downtime of live upgrades and live migrations, we focused on reducing such a delay and designed Multi-FiX.

Multi-FiX can apply complete in-placed upgrades while achieving its objective of minimizing downtime. However, Multi-FiX only protects the network availability of virtual machines thanks to two redundant NICs and bonding drivers within the hypervisor. This limitation derives from the long delay to re-initialize the PCIe controllers during the device migration (Section 5.4.4). The protection schema adopted by Multi-FiX can be generalized for stateless devices. If two independent PCIe cards that perform the same class of I/O are installed in the machines, and the software supports the transparent failover on the healthy card, then the migration schema proposed in Section 5.5.2 applies to protect the I/O workload availability during the device migration. However, stateful PCIe devices, such as disk controllers, cannot transparently failover to a healthy device (remember that RAID controllers are a single

card with multiple disks attached). The authors in [97] devised a schema for locally-attached redundant NVMe drivers to make them capable of surviving the controller reset. Nevertheless, this approach cannot be generalized for other devices (e.g., GPUs).

We emphasize that the re-initialization of the PCIe device during its migration aims at building a consistent software state at the destination hypervisor, which will match the register context in the controller. This problem also affects the migration of PCI passthrough devices (Section 2.3.1). In the context of co-located hypervisors, *this limitation can be overcome by preserving the driver state in the destination hypervisor.*

PCI passthrough is a technique that allows virtual machines to control, directly, a PCI/PCIe device with minimal hypervisor interposition. Virtual machines directly read/write the MMIO PCIe registers that control the hardware peripheral (i.e., PCIe BARs). In this scenario, the device driver is hosted, with all its computational state, inside the VM. We mentioned that PCI passthrough devices are hard to checkpoint and restore. However, this is different in the context of in-place upgrades, where the hardware devices remain the same across the warm-reboot. Indeed, in this context, researchers and cloud providers have implemented a generic and transparent restoration of PCI passthrough devices [103, 85, 105, 92, 58]. Multi-FiX can be empowered with this capability, allowing PCI passthrough devices to remain available during all stages of the upgrade. During the VM migration, the driver state is preserved within the guest and still matches the relative device hardware state. This schema protects the availability of I/O workloads via PCIe passthrough during the upgrade. A similar approach to handle PCI passthrough devices can be applied to SR-IOV. Indeed, SR-IOV devices are programmed via their Physical Function (PF), usually controlled by the hypervisor. Control of the PF can be delegated to a checkpointable/restorable user-space process (e.g. a VM) which could be migrated as well during the upgrade, or a user-space driver that supports checkpoint/restore capabilities.

Bibliography

[1] Amazon EC2 Maintenance Help Page. https://aws.amazon.com/maintenance-help/. (Visited on August 2021)). (Cited on pages 15, 26 and 48.)

[2] Amazon EC2, Spot Instance Inerruptions. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-interruptions.html#interruption-reasons. (Visited on August 2021). (Cited on page 15.)

[3] Checkpoint/Restore In Userspace. https://criu.org/Main_Page. (Visited on August 2021). (Cited on pages 18, 34 and 52.)

[4] Google Compute Engine, Live migration. https://cloud.google.com/compute/docs/instances/live-migration. (Visited on August 2021). (Cited on pages 17, 48 and 94.)

[5] Google Compute Engine, Nested virtualization overview. https://cloud.google.com/compute/docs/instances/nested-virtualization/overview. (Visited on August 2021). (Cited on pages 75 and 77.)

[6] Google Compute Engine, Preemptible VM Instances. https://cloud.google.com/compute/docs/instances/preemptible. (Visited on August 2021). (Cited on pages 15 and 48.)

[7] Intel Processor Microcode Package for Linux. https://github.com/intel/Intel-Linux-Processor-Microcode-Data-Files. (Visited on August 2021). (Cited on page 69.)

[8] Intel® Software Guard Extensions (Intel® SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html. (Visited on August 2021). (Cited on page 69.)

[9] Kpatch: Dynamic Kernel Patching, GitHub Repository. https://github.com/dynup/kpatch. (Visited on August 2021). (Cited on pages 25 and 48.)

[10] KVM, CPU Hotplug. https://www.linux-kvm.org/page/CPUHotPlug. (Visited on August 2021). (Cited on page 22.)

[11] Open vSwitch. http://openvswitch.org/. (Visited on August 2021). (Cited on page 25.)

[12] Open vSwitch, Bonding. https://docs.openvswitch.org/en/latest/topics/bonding/. (Visited on August 2021). (Cited on pages 18 and 88.)

[13] Openstack web page. https://www.openstack.org/. (Visited on August 2021). (Cited on page 104.)

[14] Physical Memory Model. https://www.kernel.org/doc/html/latest/vm/memory-model.html. (Visited on August 2021). (Cited on page 60.)

[15] QEMU Migration Documentation. https://github.com/qemu/qemu/blob/master/docs/devel/migration.rst. (Visited on August 2021). (Cited on page 54.)

[16] Redis, Web Page. https://redis.io/. (Visited on August 2021). (Cited on pages 64, 94 and 96.)

[17] SUSE Linux Enterprise Live Patching. https://www.suse.com/products/live-patching/. (Visited on August 2021). (Cited on page 25.)

[18] The Linux x86 Boot Protocol. https://www.kernel.org/doc/Documentation/x86/boot.txt. (Visited on August 2021). (Cited on pages 59, 78 and 86.)

[19] Virtuozzo Hybrid Server. https://www.virtuozzo.com/virtuozzo-hybrid-server/. (Visited on August 2021). (Cited on page 18.)

[20] VMware vSphere, Enable CPU Hot Add . https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.vm_admin.doc/GUID-285BB774-CE69-4477-9011-598FEF1E9ACB.html. (Visited on August 2021). (Cited on page 22.)

[21] Xen Project Software Overview. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview. (Visited on August 2021). (Cited on page 8.)

[22] Hyper-V Architecture. https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/role/hyper-v-server/architecture, October 2018. (Visited on August 2021). (Cited on page 8.)

[23] Payment Card Industry Data Security Standard, Requirements and Security Assessment Procedures Version 3.2.1. https://www.pcisecuritystandards.org/, May 2018. (Visited on August 2021). (Cited on pages 23 and 48.)

[24] CRIU, Upstream Kernel Commits. https://criu.org/Upstream_kernel_commits, May 2019. (Visited on August 2021). (Cited on page 34.)

[25] Microsoft Azure, Hypervisor Security on the Azure Fleet. https://docs.microsoft.com/en-us/azure/security/fundamentals/hypervisor, April 2021. (Visited on August 2021). (Cited on page 53.)

[26] Amit, N., Tsafrir, D., and Schuster, A. VSwapper: A Memory Swapper for Virtualized Environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS '14, pp. 349–366. (Cited on pages 21 and 22.)

[27] Anthony Liguori. Powering Next-Gen EC2 Instances: Deep Dive into the Nitro System. https://www.youtube.com/watch?v=e8DVmwj3OEs, November 2018. (Visited on August 2021). (Cited on page 9.)

[28] Arnold, J., and Kaashoek, M. F. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 4th ACM European Conference on Computer Systems* (2009), EuroSys '09, pp. 187–198. (Cited on pages 24 and 25.)

[29] Bacou, M., Todeschi, G., Tchana, A., Hagimont, D., Lepers, B., and Zwaenepoel, W. Drowsy-DC: Data center power management system. In *2019 IEEE International Parallel and Distributed Processing Symposium* (2019), IPDPS, pp. 825–834. (Cited on page 30.)

[30] Balouek, D., Carpen Amarie, A., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Pérez, C., Quesnel, F., Rohr, C., and Sarzyniec, L. Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science*, vol. 367 of *Communications in Computer and Information Science*. Springer International Publishing, 2013, pp. 3–20. (Cited on pages 39, 61 and 93.)

[31] Banerjee, I., Guo, F., Tati, K., and Venkatasubramanian, R. Memory Overcommitment in the ESX Server. *VMware Technical Journal 2*, 1 (2013), 2–12. (Cited on pages 22 and 23.)

[32] Barbalace, A., Ravindran, B., and Katz, D. Popcorn: a Replicated-kernel OS Based on Linux. In *Proceedings of the Linux Symposium 2014* (2014), OLS '14, pp. 123–138. (Cited on pages 70, 75, 77 and 78.)

[33] Barker, S., Wood, T., Shenoy, P., and Sitaraman, R. An Empirical Study of Memory Sharing in Virtual Machines. In *2012 USENIX Annual Technical Conference* (2012), ATC '12, pp. 273–284. (Cited on pages 20, 23 and 28.)

[34] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP '09, pp. 29–44. (Cited on pages 75 and 77.)

[35] Ben-Yehuda, M., Day, M. D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., and Yassour, B.-A. The Turtles Project: Design and Implementation of Nested Virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation* (2010), OSDI '10. (Cited on pages 51, 75 and 76.)

[36] Bila, N., de Lara, E., Joshi, K., Lagar-Cavilla, H. A., Hiltunen, M., and Satyanarayanan, M. Jettison: Efficient Idle Desktop Consolidation with Partial VM Migration. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12, pp. 211–224. (Cited on page 30.)

[37] Birke, R., Podzimek, A., Chen, L. Y., and Smirni, E. State-of-the-Practice in Data Center Virtualization: Toward a Better Understanding of VM Usage. In *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2013), DSN '13, pp. 1–12. (Cited on pages 19, 20, 28 and 48.)

[38] Breitgand, D., Dubitzky, Z., Epstein, A., Feder, O., Glikson, A., Shapira, I., and Toffetti, G. An Adaptive Utilization Accelerator for Virtualized Environments. In *IEEE International Conference on Cloud Engineering* (2014), pp. 165–174. (Cited on page 20.)

[39] Brown, L., Keshavamurthy, A., Shaohua Li, D., Moore, R., Pallipadi, V., and Luming, Y. ACPI in Linux. In *Proceedings of the Linux Symposium 2005* (2005), OLS '15, pp. 51–67. (Cited on page 78.)

[40] Bugnion, E., Nieh, J., Tsafrir, D., and Martonosi, M. *Hardware and Software Support for Virtualization*. Morgan & Claypool, 2017. (Cited on page 8.)

[41] Bui, B., Mvondo, D., Teabe, B., Jiokeng, K., Wapet, L., Tchana, A., Thomas, G., Hagimont, D., Muller, G., and De-

Palma, N. When EXtended Para-Virtualization (XPV) Meets NUMA. In *Proceedings of the Fourteenth EuroSys Conference* (2019), EuroSys '19, pp. 1–15. (Cited on page 48.)

[42] Chaubal, Charu. The Architecture of VMware ESXi. *VMware White Paper* (2008). (Cited on page 8.)

[43] Chen, Y., Farley, T., and Ye, N. QoS Requirements of Network Applications on the Internet. *Information Knowledge Systems Management 4*, 1 (2004), 55–76. (Cited on page 40.)

[44] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation* (2005), NSDI '05, pp. 273–286. (Cited on pages 3, 16, 17, 18 and 51.)

[45] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC '10, pp. 143–154. (Cited on pages 64, 94 and 96.)

[46] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, pp. 153–167. (Cited on page 20.)

[47] Depoutovitch, A., and Stumm, M. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the 5th European Conference on Computer Systems* (2010), EuroSys '10, pp. 181–194. (Cited on page 52.)

[48] Doddamani, S., Sinha, P., Lu, H., Cheng, T.-H. K., Bagdi, H. H., and Gopalan, K. Fast and Live Hypervisor Replacement. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2019), VEE '19, pp. 45–58. (Cited on pages 50, 51, 75, 76 and 77.)

[49] Fernando Luis Vázquez Cao. Generating a White List for Hardware which Works with Kexec/Kdump. In *LinuxConf Europe 2007* (2007). (Cited on pages 51 and 74.)

[50] Giuffrida, C., et al. Safe and automatic live update. *VU University Amsterdam* (2014). (Cited on page 24.)

[51] Giuffrida, C., Iorgulescu, C., Kuijsten, A., and Tanenbaum, A. S. Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer. In *Lucky LISA: Proceedings of the 27th Large Installation System Administration Conference, LISA 2013*. (Cited on page 48.)

[52] Goel, A., Chopra, B., Gerea, C., Mátáni, D., Metzler, J., Ul Haq, F., and Wiener, J. Fast Database Restarts at Facebook. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014), SIGMOD '14, pp. 541–549. (Cited on pages 16, 24, 48 and 67.)

[53] Golden, B. Inside Amazon's Cloud: Just How Many Customer Projects? https://www.cio.com/article/2424378/virtualization/inside-amazon-s-cloud--just-how-many-customer-projects-.html, September 2009. (Visited on August 2021). (Cited on page 43.)

[54] Goyal, V., Biederman, E. W., and Nellitheertha, H. Kdump, A Kexec-based Kernel Crash Dumping Mechanism . In *Proceedings of the Linux Symposium 2005* (2005), OLS '05, pp. 169–180. (Cited on page 92.)

[55] Hermenier, F., Lawall, J. L., and Muller, G. Btrplace: A flexible consolidation manager for highly available applications. *IEEE Trans. Dependable Sec. Comput. 10*, 5 (2013), 273–286. (Cited on page 104.)

[56] Hermenier, F., Ramesh, A., Nagpal, A., Shukla, H., and Chandra, R. Hotspot Mitigations for the Masses. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), SoCC '19, pp. 102–113. (Cited on pages 9, 19, 20 and 28.)

[57] Hines, M. R., Deshpande, U., and Gopalan, K. Post-Copy Live Migration of Virtual Machines. *SIGOPS Operating Systems Review 43*, 3 (2009), 14–26. (Cited on pages 17 and 51.)

[58] Jason Zeng. KVM Forum, Device Keepalive State For Local Live Migration and VMM Fast Restart. https://kvmforum2020.sched.com/event/eE3W/device-keepalive-state-for-local-live-migration-and-vmm-fast-restart-jason-zeng-intel, October 2020. (Cited on page 105.)

[59] Jonathan Corbet. Kexec. https://lwn.net/Articles/15468/. (Visited on August 2021). (Cited on pages 52, 58 and 74.)

[60] Joshi, A., Pimpale, S., Naik, M., Rathi, S., and Pawar, K. TwinLinux: Running independent Linux Kernels simultaneously on separate cores of a multicore system. In *Proceedings of the Linux Symposium 2010* (2010), OLS '10, pp. 101–108. (Cited on pages 77 and 78.)

[61] Kashyap, S., Min, C., Lee, B., Kim, T., and Emelyanov, P. Instant OS Updates via Userspace Checkpoint-and-Restart. In *2016 USENIX Annual Technical Conference* (2016), ATC' 16, pp. 605–619. (Cited on pages 3 and 52.)

[62] Kherbache, V., Madelaine, E., and Hermenier, F. Planning Live-Migrations to Prepare Servers for Maintenance. In *Euro-Par 2014: Parallel Processing Workshops* (2014), Springer, pp. 498–507. (Cited on pages 20 and 23.)

[63] Kim, J., Ruggiero, M., and Atienza, D. Free cooling-aware dynamic power management for green datacenters. In *2012 International Conference on High Performance Computing Simulation* (2012), HPCS 2012, pp. 140–146. (Cited on page 19.)

[64] Knauth, T., and Fetzer, C. DreamServer: Truly On-Demand Cloud Services. In *Proceedings of International Conference on Systems and Storage* (2014), SYSTOR 2014, pp. 1–11. (Cited on pages 3, 28 and 31.)

[65] Koomey, J., and Taylor, J. Zombie/Comatose Servers Redux. https://www.anthesisgroup.com/report-zombie-and-comatose-servers-redux-jon-taylor-and-jonathan-koomey/, April 2017. (Visited on August 2021). (Cited on pages 2 and 28.)

[66] Kourai, K., and Chiba, S. Fast Software Rejuvenation of Virtual Machine Monitors. *IEEE Transactions on Dependable and Secure Computing 8*, 6 (2011), 839–851. (Cited on pages 52 and 74.)

[67] Kourai, K., and Ooba, H. Zero-Copy Migration for Lightweight Software Rejuvenation of Virtualized Systems. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (2015), APSys '15. (Cited on pages 50 and 76.)

[68] Lai Jiangshan. QEMU Patch, Add Capability to Bypass the Shared Memory. https://lists.gnu.org/archive/html/qemu-devel/2018-04/msg02250.html, April 2018. (Visited on August 2021). (Cited on page 59.)

[69] Le, M., and Tamir, Y. ReHype: Enabling VM Survival across Hypervisor Failures. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2011), VEE '11, pp. 63–74. (Cited on page 52.)

[70] Lottiaux, R., Boissinot, B., Gallard, P., Vallée, G., and Morin, C. OpenMosix, OpenSSI and Kerrighed: a comparative study. In *IEEE International Symposium on Cluster Computing and the Grid* (2005), CCGRID '05, pp. 1016–1023. (Cited on page 18.)

[71] Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., and Huici, F. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, pp. 218–233. (Cited on page 14.)

[72] Mike Rapoport. A Quick History of Early-Boot Memory Allocators. https://lwn.net/Articles/761215/, July 2018. (Visited on August 2021). (Cited on page 53.)

[73] Mike Rapoport. Four-level page tables. https://lwn.net/Articles/106177/, October 2018. (Visited on August 2021). (Cited on page 60.)

[74] Nitu, V., Teabe, B., Tchana, A., Isci, C., and Hagimont, D. Welcome to Zombieland: Practical and Energy-Efficient Memory Disaggregation in a Datacenter. In *Proceedings of the Thirteenth EuroSys Conference* (2018), EuroSys '18, pp. 1–12. (Cited on pages 28 and 30.)

[75] Nomura, Y., Senzaki, R., Nakahara, D., Ushio, H., Kataoka, T., and Taniguchi, H. Mint: Booting Multiple Linux Kernels on a Multicore Processor. In *2011 International Conference on Broadband and Wireless Computing, Communication and Applications* (2011), pp. 555–560. (Cited on pages 77 and 78.)

[76] Pan, Z., Dong, Y., Chen, Y., Zhang, L., and Zhang, Z. CompSC: Live Migration with Pass-through Devices. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments* (2012), VEE '12, pp. 109–120. (Cited on page 17.)

[77] Panwar, A., Prasad, A., and Gopinath, K. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018), ASPLOS '18:, pp. 679–692. (Cited on page 48.)

[78] Popek, G. J., and Goldberg, R. P. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM 17*, 7 (1974), 412–421. (Cited on page 9.)

[79] Potter, S., and Nieh, J. Reducing Downtime Due to System Maintenance and Upgrades. In *Proceedings of the 19th Conference on Systems Administration* (2015), LISA 2005. (Cited on page 52.)

[80] Prasad, A., Gopinath, K., and McKenney, P. E. The RCU-Reader Preemption Problem in VMs. In *2017 USENIX Annual Technical Conference* (2017), ATC '17, pp. 265–270. (Cited on page 21.)

[81] Reich, J., Goraczko, M., Kansal, A., and Padhye, J. Sleepless in Seattle No Longer. In *2010 USENIX Annual Technical Conference* (ATC '10, 2010). (Cited on pages 28 and 30.)

[82] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing* (2012), SoCC '12. (Cited on pages 2, 20 and 28.)

[83] Rizzo, L. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference* (2012), ATC '12, pp. 101–112. (Cited on page 77.)

[84] Ruprecht, A., Jones, D., Shiraev, D., Harmon, G., Spivak, M., Krebs, M., Baker-Harvey, M., and Sanderson, T. VM Live Migration At Scale. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '18, pp. 45–56. (Cited on pages 2, 3, 16, 17, 20, 23, 48 and 51.)

[85] Russinovich, M., Govindaraju, N., Raghuraman, M., Hepkin, D., Schwartz, J., and Kishan, A. Virtual machine preserving host updates for zero day patching in public cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021), EuroSys '21, pp. 114–129. (Cited on pages 17, 18, 24, 50, 52, 53, 74 and 105.)

[86] Schopp, J., Hansen, D., Kravetz, M., Takahashi, H., Iwamoto, T., Goto, Y., Kamezawa, H., Tolentino, M., and Picco, B. Hotplug Memory Redux. In *Proceedings of the Linux Symposium 2005* (2005), OLS '05, pp. 152–174. (Cited on page 60.)

[87] Segalini, A., Lopez-Pacheco, D., Urvoy-Keller, G., Hermenier, F., and Jaquemart, Q. Hy-FiX: Fast In-place Upgrades of KVM Hypervisors. *IEEE Transactions on Cloud Computing* (2021). Early Access. (Cited on pages 16, 51, 53 and 74.)

[88] Segalini, A., Pacheco, D. L., Urvoy-Keller, G., Hermenier, F., and Jacquemart, Q. Hy-fix: Fast in-place upgrade of kvm hypervisors. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), SoCC '19, pp. 485–485. (Cited on page 49.)

[89] Sharma, P., Chaufournier, L., Shenoy, P., and Tay, Y. Containers and Virtual Machines at Scale: A Comparative Study. In *Pro-*

ceedings of the 17th International Middleware Conference (2016), Middleware '16, pp. 1–13. (Cited on pages 14, 19 and 21.)

[90] Shi, B., and Shen, H. Memory/Disk Operation Aware Lightweight VM Live Migration Across Data-centers with Low Performance Impact. In *2019 IEEE Conference on Computer Communications* (2019), INFOCOM 2019, pp. 334–342. (Cited on pages 26 and 51.)

[91] Soundararajan, V., and Anderson, J. M. The Impact of Management Operations on the Virtualized Datacenter. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010), pp. 326–337. (Cited on page 23.)

[92] Steven Sistare. KVM Forum, QEMU Live Update. https://kvmforum2020.sched.com/event/eE3E, October 2020. (Cited on page 105.)

[93] Sugerman, J., Venkitachalam, G., and Lim, B.-H. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *2001 USENIX Annual Technical Conference* (2001), ATC '01, pp. 1–14. (Cited on page 9.)

[94] Tanenbaum, A. S., and Bos, H. *Modern Operating Systems*. Pearson, 2015. (Cited on page 21.)

[95] Taylor Brown. Bringing Docker To Windows Developers with Windows Server Containers . https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/april/containers-bringing-docker-to-windows-developers-with-windows-server-containers, April 2017. (Visited on August 2021). (Cited on page 14.)

[96] Verma, A., Bagrodia, J., and Jaiswal, V. Virtual Machine Consolidation in the Wild. In *Proceedings of the 15th International Middleware Conference* (2014), Middleware '14, pp. 313–324. (Cited on pages 19 and 20.)

[97] Xue, S., Zhao, S., Chen, Q., Deng, G., Liu, Z., Zhang, J., Song, Z., Ma, T., Yang, Y., Zhou, Y., et al. Spool: Reliable Virtualized NVMe Storage Pool in Public Cloud Infrastructure. In *2020 USENIX Annual Technical Conference* (2020), ATC '20, pp. 97–110. (Cited on pages 98 and 105.)

[98] Yamada, H., and Kono, K. Traveling Forward in Time to Newer Operating Systems Using ShadowReboot. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2013), VEE '13, pp. 121–130. (Cited on page 52.)

[99] Zellweger, G., Gerber, S., Kourtis, K., and Roscoe, T. Decoupling Cores, Kernels, and Operating Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI '14, pp. 17–31. (Cited on page 77.)

[100] Zhai, E., Cummings, G. D., and Dong, Y. Live Migration with Pass-through Device for Linux VM. In *Proceedings of the Linux Symposium 2005* (2005), OLS '15, pp. 51–67. (Cited on pages 17 and 70.)

[101] Zhang, I., Denniston, T., Baskakov, Y., and Garthwaite, A. Optimizing vm checkpointing for restore performance in vmware esxi. (Cited on page 38.)

[102] Zhang, L., Litton, J., Cangialosi, F., Benson, T., Levin, D., and Mislove, A. Picocenter: Supporting Long-Lived, Mostly-Idle Applications in Cloud Environments. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys '16. (Cited on pages 2, 27, 28 and 31.)

[103] Zhang, X., Zheng, X., Wang, Z., Li, Q., Fu, J., Zhang, Y., and Shen, Y. 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* (2019), ASPLOS '19, pp. 93–105. (Cited on pages 3, 25, 48, 50, 51 and 105.)

[104] Zhi, J., Bila, N., and de Lara, E. Oasis: Energy Proportionality with Hybrid Server Consolidation. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys '16. (Cited on pages 2, 28, 30 and 44.)

[105] Zhimin Feng. KVM Forum, KVM Live Upgrade With Properly Handling Of Passthrough Devices. https://kvmforum2020.sched.com/event/eE3c/kvm-live-upgrade-with-properly-handling-of-passthrough-devices-zhimin-feng-bytedance, October 2020. (Visited on August 2021). (Cited on page 105.)

[106] Zhou, D., and Tamir, Y. Fast Hypervisor Recovery Without Reboot. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2018), DSN '18, pp. 115–126. (Cited on page 52.)