Hardware and Software Support for Virtualization

Synthesis Lectures on Computer Architecture

Editor
Margaret Martonosi, *Princeton University*

Synthesis Lectures on Computer Architecture publishes 50- to 100-page publications on topics pertaining to the science and art of designing, analyzing, selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals. The scope will largely follow the purview of premier computer architecture conferences, such as ISCA, HPCA, MICRO, and ASPLOS.

Hardware and Software Support for Virtualization
Edouard Bugnion, Jason Nieh, and Dan Tsafrir
2017

Datacenter Design and Management: A Computer Architect's Perspective
Benjamin C. Lee
2016

A Primer on Compression in the Memory Hierarchy
Somayeh Sardashti, Angelos Arelakis, Per Stenström, and David A. Wood
2015

Research Infrastructures for Hardware Accelerators
Yakun Sophia Shao and David Brooks
2015

Analyzing Analytics
Rajesh Bordawekar, Bob Blainey, and Ruchir Puri
2015

Customizable Computing
Yu-Ting Chen, Jason Cong, Michael Gill, Glenn Reinman, and Bingjun Xiao
2015

Die-stacking Architecture
Yuan Xie and Jishen Zhao
2015

Single-Instruction Multiple-Data Execution
Christopher J. Hughes
2015

Power-Efficient Computer Architectures: Recent Advances
Magnus Själander, Margaret Martonosi, and Stefanos Kaxiras
2014

FPGA-Accelerated Simulation of Computer Systems
Hari Angepat, Derek Chiou, Eric S. Chung, and James C. Hoe
2014

A Primer on Hardware Prefetching
Babak Falsafi and Thomas F. Wenisch
2014

On-Chip Photonic Interconnects: A Computer Architect's Perspective
Christopher J. Nitta, Matthew K. Farrens, and Venkatesh Akella
2013

Optimization and Mathematical Modeling in Computer Architecture
Tony Nowatzki, Michael Ferris, Karthikeyan Sankaralingam, Cristian Estan, Nilay Vaish, and David Wood
2013

Security Basics for Computer Architects
Ruby B. Lee
2013

The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition
Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle
2013

Shared-Memory Synchronization
Michael L. Scott
2013

Resilient Architecture Design for Voltage Variation
Vijay Janapa Reddi and Meeta Sharma Gupta
2013

Multithreading Architecture
Mario Nemirovsky and Dean M. Tullsen
2013

Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)
Hyesoon Kim, Richard Vuduc, Sara Baghsorkhi, Jee Choi, and Wen-mei Hwu
2012

Automatic Parallelization: An Overview of Fundamental Compiler Techniques
Samuel P. Midkiff
2012

Phase Change Memory: From Devices to Systems
Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran
2011

Multi-Core Cache Hierarchies
Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar
2011

A Primer on Memory Consistency and Cache Coherence
Daniel J. Sorin, Mark D. Hill, and David A. Wood
2011

Dynamic Binary Modification: Tools, Techniques, and Applications
Kim Hazelwood
2011

Quantum Computing for Computer Architects, Second Edition
Tzvetan S. Metodi, Arvin I. Faruque, and Frederic T. Chong
2011

High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities
Dennis Abts and John Kim
2011

Processor Microarchitecture: An Implementation Perspective
Antonio González, Fernando Latorre, and Grigorios Magklis
2010

Transactional Memory, 2nd edition
Tim Harris, James Larus, and Ravi Rajwar
2010

Computer Architecture Performance Evaluation Methods
Lieven Eeckhout
2010

Introduction to Reconfigurable Supercomputing
Marco Lanzagorta, Stephen Bique, and Robert Rosenberg
2009

On-Chip Networks
Natalie Enright Jerger and Li-Shiuan Peh
2009

The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It
Bruce Jacob
2009

Fault Tolerant Computer Architecture
Daniel J. Sorin
2009

The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines
Luiz André Barroso and Urs Hölzle
2009

Computer Architecture Techniques for Power-Efficiency
Stefanos Kaxiras and Margaret Martonosi
2008

Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency
Kunle Olukotun, Lance Hammond, and James Laudon
2007

Transactional Memory
James R. Larus and Ravi Rajwar
2006

Quantum Computing for Computer Architects
Tzvetan S. Metodi and Frederic T. Chong
2006

Hardware and Software Support for Virtualization

Edouard Bugnion, Jason Nieh, and Dan Tsafrir

www.morganclaypool.com

ISBN: 9781627056939 paperback
ISBN: 9781627056885 ebook

DOI 10.2200/S00754ED1V01Y201701CAC038

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Lecture #38
Series Editor: Margaret Martonosi, *Princeton University*
Series ISSN
Print 1935-3235 Electronic 1935-3243

Hardware and Software Support for Virtualization

Edouard Bugnion
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Jason Nieh
Columbia University

Dan Tsafrir
Technion – Israel Institute of Technology

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #38

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

This book focuses on the core question of the necessary *architectural support provided by hardware* to efficiently run virtual machines, and of the corresponding design of the *hypervisors* that run them. Virtualization is still possible when the instruction set architecture lacks such support, but the hypervisor remains more complex and must rely on additional techniques.

Despite the focus on architectural support in current architectures, some historical perspective is necessary to appropriately frame the problem. The first half of the book provides the historical perspective of the theoretical framework developed four decades ago by Popek and Goldberg. It also describes earlier systems that enabled virtualization despite the lack of architectural support in hardware.

As is often the case, theory defines a necessary—but not sufficient—set of features, and modern architectures are the result of the combination of the theoretical framework with insights derived from practical systems. The second half of the book describes state-of-the-art support for virtualization in both x86-64 and ARM processors. This book includes an in-depth description of the CPU, memory, and I/O virtualization of these two processor architectures, as well as case studies on the Linux/KVM, VMware, and Xen hypervisors. It concludes with a performance comparison of virtualization on current-generation x86- and ARM-based systems across multiple hypervisors.

KEYWORDS

computer architecture, virtualization, virtual machine, hypervisor, dynamic binary translation

Contents

Preface

> "Virtual machines have finally arrived. Dismissed for a number of years as merely academic curiosities, they are now seen as cost-effective techniques for organizing computer systems resources to provide extraordinary system flexibility and support for certain unique applications".
>
> Robert. P. Goldberg, *IEEE Computer*, 1974 [78]

The academic discipline of computer systems research, including computer architecture, is in many aspects more tidal than linear: specific ingrained, well-understood techniques lose their relevance as tradeoffs evolve. Hence, the understanding of these techniques then ebbs from the collective knowledge of the community. Should the architectural tide later flow in the reverse direction, we have the opportunity to reinvent—or at least appreciate once more—old concepts all over again.

The history of virtualization is an excellent example of this cycle of innovation. The approach was popular in the early era of computing, as demonstrated from the opening quote. At high tide in the 1970s, hundreds of papers were written on virtualization with conferences and workshops dedicated to the topic. The era established the basic principles of virtualization and entire compute stacks—hardware, virtual machine monitors, and operating systems—were designed to efficiently support virtual machines. However, the tide receded quickly in the early 1980s as operating systems matured; virtual machines were soon strategically discarded in favor of a more operating system-centric approach to building systems.

Throughout the 1980s and 1990s, with the appearance of the personal computer and client/server era, virtual machines were largely relegated to a mainframe-specific curiosity. For example, the processors developed in that era (MIPS, Sparc, x86), were not explicitly designed to provide architectural support for virtualization, since there was no obvious business requirement to maintain support for virtual machines. In addition, and in good part because of the ebb of knowledge of the formal requirements for virtualization, many of these architectures made arbitrary design decisions that violated the basic principles established a decade earlier.

For most computer systems researchers of the open systems era, raised on UNIX, RISC, and x86, virtual machines were perceived to be just another bad idea from the 1970s. In 1997, the Disco [44] paper revisited virtual machines with a fresh outlook, specifically as the founda-

tion to run commodity operating systems on scalable multiprocessors. In 1999, VMware released VMware Workstation 1.0 [45], the first commercial virtualization solution for x86 processors.

At the time, researchers and commercial entities started building virtual machines solutions for desktops and servers. A few years later, the approach was introduced to mobile platforms. Disco, VMware Workstation, VMware ESX Server [177], VirtualPC [130], Xen [27], Denali [182], and Cells [16], were all originally designed for architectures that did *not* provide support for virtualization. These different software systems each took a different approach to work around the limitations of the hardware of the time. Although processor architectures have evolved to provide hardware support for virtualization, many of the key innovations of that era such as hosted architectures [162], paravirtualization [27, 182], live migration [51, 135], and memory ballooning [177], remain relevant today, and have a profound impact on computer architecture trends.

Clearly, the virtualization tide has turned, to the point that it is once more a central driver of innovation throughout the industry, including system software, systems management, processor design, and I/O architectures. As a matter of fact, the exact quote from Goldberg's 1974 paper would have been equally timely 30 years later: Intel introduced its first-generation hardware support for virtual machines in 2004. Every maintained virtualization solution, including VMware Workstation, ESX Server, and Xen, quickly evolved to leverage the benefits of hardware support for virtualization. New systems were introduced that assumed the existence of such hardware support as a core design principle, notably KVM [113]. With the combined innovation in hardware and software and the full support of the entire industry, virtual machines quickly became central to IT organizations, where they were used among other things to improve IT efficiency, simplify provisioning, and increase availability of applications. Virtual machines were also proposed to uniquely solve hard open research questions, in domains such as live migration [51, 135] and security [73]. Within a few years, they would play a central role in enterprise datacenters. For example, according to the market research firm IDC, since 2009 there are more virtual machines deployed than physical hosts [95].

Today, virtual machines are ubiquitous in enterprise environments, where they are used to virtualize servers as well as desktops. They form the foundation of all Infrastructure-as-a-Service (IAAS) clouds, including Amazon EC2, Google CGE, Microsoft Azure, and OpenStack. Once again, the academic community dedicates conference tracks, sessions, and workshops to the topic (e.g., the annual conference on Virtual Execution Environments (VEE)).

ORGANIZATION OF THIS BOOK

This book focuses on the core question of the necessary *architectural support provided by hardware* to efficiently run virtual machines. Despite the focus on architectural support in current architectures, some historical perspective is necessary to appropriately frame the problem. Specifically, this includes both a theoretical framework, and a description of the systems enabling virtualization despite the lack of architectural support in hardware. As is often the case, theory defines

a necessary—but not sufficient—set of features, and modern architectures are the result of the combination of the theoretical framework with insights derived from practical systems.

The book is organized as follows.

- Chapter 1 introduces the fundamental definitions of the abstraction ("virtual machines"), the run-time ("virtual machine monitors"), and the principles used to implement them.
- Chapter 2 provides the necessary theoretical framework that defines whether an instruction set architecture (ISA) is virtualizable or not, as formalized by Popek and Goldberg [143].
- Chapter 3 then describes the first set of systems designed for platforms that failed the Popek/Goldberg test. These systems each use a particular combination of workarounds to run virtual machines on platforms not designed for them. Although a historical curiosity by now, some of the techniques developed during that era remain relevant today.
- Chapter 4 focuses on the architectural support for virtualization of modern x86-64 processors, and in particular Intel's VT-x extensions. It uses KVM as a detailed case study of a hypervisor specifically designed to assume the presence of virtualization features in processors.
- Chapter 5 continues the description of x86-64 on the related question of the architectural support for MMU virtualization provided by extended page tables (also known as nested page tables).
- Chapter 6 closes the description of x86-64 virtualization with the various forms of I/O virtualization available. The chapter covers key concepts such as I/O emulation provided by hypervisors, paravirtual I/O devices, pass-through I/O with SR-IOV, IOMMUs, and the support for interrupt virtualization.
- Chapter 7 describes the architectural support for virtualization of the ARM processor family, and covers the CPU, MMU, and I/O considerations. The chapter emphasizes some of the key differences in design decisions between x86 and ARM.
- Chapter 8 compares the performance and overheads of virtualization extensions on x86 and on ARM.

In preparing this book, the authors made some deliberate decisions. First, for brevity, we focused on the examples of architectural support for virtualization, primarily around two architectures: x86-64 and ARM. Interested readers are hereby encouraged to study additional instruction set architectures. Among them, IBM POWER architecture, with its support for both hypervisor-based virtualization and logical partitioning (LPAR), is an obvious choice [76]. The SPARC architecture also provides built-in support for logical partitioning, called logical domains [163]. We also omit any detailed technical description of mainframe and mainframe-era architectures. Readers

interested in that topic should start with Goldberg's survey paper [78] and Creasy's overview of the IBM VM/370 system [54].

Second, we focused on mainstream (i.e., traditional) forms of virtual machines and the construction of hypervisors in both the presence or the absence of architectural support for virtualization in hardware. This focus is done at the expense of a description of some more advanced research concepts. For example, the text does not discuss recursive virtual machines [33, 158], the use of virtualization hardware for purposes other than running traditional virtual machines [24, 29, 31, 43, 88], or the emerging question of architectural support for containers such as Docker [129].

AUTHORS' PERSPECTIVES

This book does not attempt to cover all aspects of virtualization. Rather, it mostly focuses on the key question of the interaction between the underlying computer architecture and the systems software built on top of it. It also comes with a point of view, based on the authors' direct experiences and perspectives on the topic.

Edouard Bugnion was fortunate to be part of the Disco team as a graduate student. Because of the stigma associated with virtual machines of an earlier generation, we named our prototype in reference to the questionable musical contribution of that same decade [55], which was then coincidentally making a temporary comeback. Edouard later co-founded VMware, where he was one of the main architects and implementers of VMware Workstation, and then served as its Chief Technology Officer. In 2005, he co-founded Nuova Systems, a hardware company premised on providing architectural support for virtualization in the network and the I/O subsystem, which became the core of Cisco's Data Center strategy. More recently, having returned to academia as a professor at École polytechnique fédérale de Lausanne (EPFL), Edouard is now involved in the IX project [30, 31, 147] which leverages virtualization hardware and the Dune framework [29] to build specialized operating systems.

Jason Nieh is a Professor of Computer Science at Columbia University, where he has led a wide range of virtualization research projects that have helped shape commercial and educational practice. Zap [138], an early lightweight virtual machine architecture that supported migration, led to the development of Linux namespaces and Linux containers, as well as his later work on Cells [16, 56], one of the first mobile virtualization solutions. Virtual Layered File Systems [144, 145] introduced the core ideas of layers and repositories behind Docker and CoreOS. KVM/ARM [60] is widely deployed and used as the mainline Linux ARM hypervisor, and has led to improvements in ARM architectural support for virtualization [58]. MobiDesk [26], THINC [25], and other detailed measurement studies helped make the case for virtual desktop infrastructure, which has become widely used in industry. A dedicated teacher, Jason was the first to introduce virtualization as a pedagogical tool for teaching hands-on computer science courses, such as operating systems [136, 137], which has become common practice in universities around the world.

Dan Tsafrir is an Associate Professor at the Technion—Israel Institute of Technology, where he regularly appreciates how fortunate he is to be working with brilliant students on cool projects for a living. Some of these projects drive state-of-the-art virtualization forward. For example, vIOMMU showed for the first time how to fully virtualize I/O devices on separate (side)cores without the knowledge or involvement of virtual machines, thus eliminating seemingly inherent trap-and-emulate virtualization overheads [12]. vRIO showed that such sidecores can in fact be consolidated on separate remote servers, enabling a new kind of datacenter-scale I/O virtualization model that is cheaper and more performant than existing alternatives [116]. ELI introduced software-based exitless interrupts—a concept recently adopted by hardware—which, after years of efforts, finally provided bare-metal performance for high-throughput virtualization workloads [13, 80]. VSwapper showed that uncooperative swapping of memory of virtual machines can be made efficient, despite the common belief that this is impossible [14]. Virtual CPU validation showed how to uncover a massive amount of (confirmed and now fixed) hypervisor bugs by applying Intel's physical CPU testing infrastructure to the KVM hypervisor [15]. EIOVAR and its successor projects allowed for substantially faster and safer IOMMU protection and found their way into the Linux kernel [126, 127, 142]. NPFs provide page-fault support for network controllers and are now implemented in production Mellanox NICs [120].

TARGET AUDIENCE

This book is written for researchers and graduate students who have already taken a basic course in both computer architecture and operating systems, and who are interested in becoming fluent with virtualization concepts. Given the recurrence of virtualization in the literature, it should be particularly useful to new graduate students before they start reading the many papers treating a particular sub-aspect of virtualization. We include numerous references of widely read papers on the topic, together with a high-level, modern commentary on their impact and relevance today.

Edouard Bugnion, Jason Nieh, and Dan Tsafrir
January 2017

Acknowledgments

This book would not have happened without the support of many colleagues. The process would have not even started without the original suggestion from Rich Uhlig to Margaret Martonosi, the series editor. The process, in all likelihood, would have never ended without the constant, gentle probing of Mike Morgan; we thank him for his patience. Ole Agesen, Christoffer Dall, Arthur Kiyanovski, Shih-Wei Li, Jintack Lim, George Prekas, Jeff Sheldon, and Igor Smolyar provided performance figures specifically for this book; students will find the additional quantitative data enlightening. Margaret Church made multiple copy-editing passes to the manuscript; we thank her for the diligent and detailed feedback at each round. Nadav Amit, Christoffer Dall, Nathan Dauthenhahn, Canturk Isci, Arthur Kiyanovski, Christos Kozyrakis, Igor Smolyar, Ravi Soundararajan, Michael Swift, and Idan Yaniv all provided great technical feedback on the manuscript.

The authors would like to thank EPFL, Columbia University, and the Technion—Israel Institute of Technology, for their institutional support. Bugnion's research group is supported in part by grants from Nano-Tera, the Microsoft EPFL Joint Research Center, a Google Graduate Research Fellowship and a VMware research grant. Nieh's research group is supported in part by ARM Ltd., Huawei Technologies, a Google Research Award, and NSF grants CNS-1162447, CNS-1422909, and CCF-1162021. Tsafrir's research group is supported in part by research awards from Google Inc., Intel Corporation, Mellanox Technologies, and VMware Inc., as well as by funding from the Israel Science Foundation (ISF) grant No. 605/12, the Israeli Ministry of Economics via the HIPER consortium, the joint BSF-NSF United States-Israel Binational Science Foundation and National Science Foundation grant No. 2014621, and the European Union's Horizon 2020 research and innovation programme grant agreement No. 688386 (OPERA).

Edouard Bugnion, Jason Nieh, and Dan Tsafrir
Lausanne, New York, and Haifa
January 2017

CHAPTER 1

Definitions

This chapter introduces the basic concepts of virtualization, virtual machines, and virtual machine monitors. This is necessary for clarity as various articles, textbooks, and commercial product descriptions sometimes use conflicting definitions. We use the following definitions in this book.

- **Virtualization** is the application of the layering principle through enforced modularity, whereby the exposed virtual resource is identical to the underlying physical resource being virtualized.
- A **virtual machine** is an abstraction of a complete compute environment through the combined virtualization of the processor, memory, and I/O components of a computer.
- The **hypervisor** is a specialized piece of system software that manages and runs virtual machines.
- The **virtual machine monitor** (**VMM**) refers to the portion of the hypervisor that focuses on the CPU and memory virtualization.[1]

The rest of this chapter is organized as follows. We formalize the definitions of virtualization, virtual machines, and hypervisors in §1.1, §1.2, and §1.3, respectively. §1.4 classifies existing hypervisors into type-1 (bare-metal) and type-2 (hosted) architectures. We then provide a sketch illustration of a hypervisor in §1.5 while deferring the formal definition until Chapter 2. In addition to the three basic concepts, we also define useful, adjacent concepts such as the different terms for memory (§1.6) and various approaches to virtualization and paravirtualization (§1.7). We conclude in §1.8 with a short description of some of the key reasons why virtual machines play a fundamental role in information technology today.

1.1 VIRTUALIZATION

Virtualization is the application of the layering principle through enforced modularity, whereby the exposed virtual resource is identical to the underlying physical resource being virtualized.

[1] N.B.: the terms hypervisor and virtual machine monitor have been used interchangeably in the literature. Here, we prefer the term *hypervisor* when describing an entire system and the term *VMM* when describing the subsystem that virtualizes the CPU and memory, or in its historical formal context in Chapter 2.

This definition is grounded in two fundamental principles of computer systems. First, **layering** is the presentation of a single abstraction, realized by adding a level of indirection, when (i) the indirection relies on a single lower layer and (ii) uses a well-defined namespace to expose the abstraction. Second, **enforced modularity** additionally guarantees that the clients of the layer cannot bypass the abstraction layer, for example to access the physical resource directly or have visibility into the usage of the underlying physical namespace. Virtualization is therefore nothing more than an instance of layering for which the exposed abstraction is equivalent to the underlying physical resource.

This combination of indirection, enforced modularity, and compatibility is a particularly powerful way to both reduce the complexity of computer systems and simplify operations. Let's take the classic example of RAID [48], in which a redundant array of inexpensive disk is aggregated to form a single, virtual disk. Because the interface is compatible (it is a block device for both the virtual and physical disks), a filesystem can be deployed identically, whether the RAID layer is present or not. As the RAID layer manages its own resources internally, hiding the physical addresses from the abstraction, physical disks can be swapped into the virtual disk transparently from the filesystem using it; this simplifies operations, in particular when disks fail and must be replaced. Even though RAID hides many details of the organization of the storage subsystem from the filesystem, the operational benefits clearly outweigh any potential drawbacks resulting from the added level of indirection.

As broadly defined, virtualization is therefore not synonymous to virtual machines. It is also not limited to any particular field of computer science or location in the compute stack. In fact, virtualization is prevalent across domains. We provide a few examples found in hardware, software, and firmware.

Virtualization in Computer Architecture: Virtualization is obviously a fundamental part of computer architecture. Virtual memory, as exposed through memory management units (MMU), serves as the canonical example: the MMU adds a level of indirection which hides the physical addresses from applications, in general through a combination of segmentation and paging mechanisms. This enforces modularity as MMU control operations are restricted to kernel mode. As both physical memory and virtual memory expose the same abstraction of byte-addressable memory, the same instruction set architecture can operate identically with virtual memory when the MMU is enabled, and with physical memory when it is disabled.

Virtualization within Operating Systems: Operating systems have largely adopted the same concept. In fact, at its core, an operating system does little more than safely expose the resources of a computer—CPU, memory, and I/O—to multiple, concurrent applications. For example, an operating system controls the MMU to expose the abstraction of isolated address spaces to processes; it schedules threads on the physical cores transparently, thereby multiplexing in software the limited physical CPU resource; it mounts multiple distinct filesystems into a single virtualized namespace.

Virtualization in I/O subsystems: Virtualization is ubiquitous in disks and disk controllers, where the resource being virtualized is a block-addressed array of sectors. The approach is used by RAID controllers and storage arrays, which present the abstraction of multiple (virtual) disks to the operating systems, which addresses them as (real) disks. Similarly, the Flash Translation Layer found in current SSD provides wear-leveling within the I/O subsystem and exposes the SSD to the operating systems as though it were a mechanical disk.

Whether done in hardware, in software, or embedded in subsystems, virtualization is always achieved by using and combining three simple techniques, illustrated in Figure 1.1. First, **multiplexing** exposes a resource among multiple virtual entities. There are two types of multiplexing, in space and in time. With space multiplexing, the physical resource is partitioned (in space) into virtual entities. For example, the operating system multiplexes different pages of physical memory across different address spaces. To achieve this goal, the operating system manages the virtual-to-physical mappings and relies on the architectural support provided by the MMU.

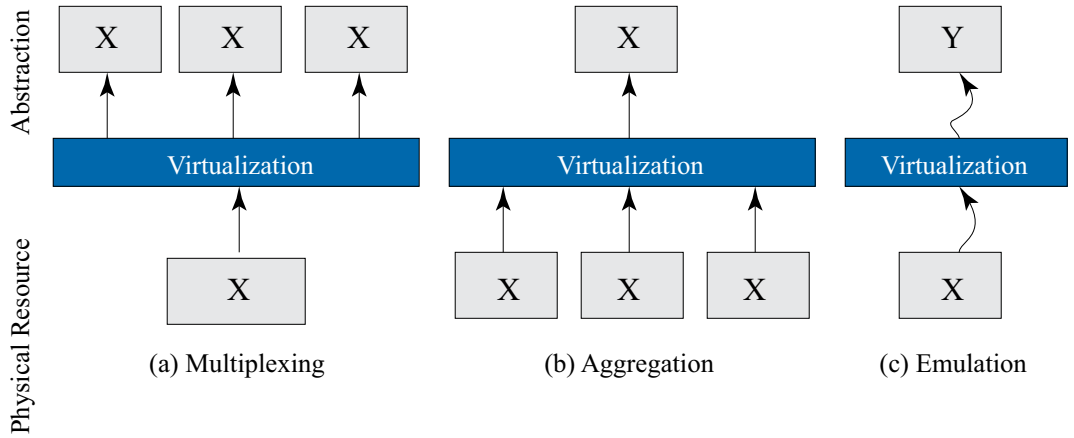

Figure 1.1: Three basic implementations techniques of virtualization. X represents both the physical resource and the virtualized abstraction.

With time multiplexing, the same physical resource is scheduled temporally between virtual entities. For example, the OS scheduler multiplexes the CPU core and hardware threads among the set of runnable processes. The context switching operation saves the processor's register file in the memory associated with the outgoing process, and then restores the state of the register file from the memory location associated with the incoming process.

Second, **aggregation** does the opposite, it takes multiple physical resources and makes them appear as a single abstraction. For example, a RAID controller aggregates multiple disks into a single volume. Once configured, the controller ensures that all read and write operations to the volume are appropriately reflected onto the various disks of the RAID group. The operating system then formats the filesystem onto the volume without having to worry about the details of

the layout and the encoding. In a different domain, a processor's memory controller aggregates the capacity of multiple DIMMs into a single physical address space, which is then managed by the operating system.

Third, **emulation** relies on a level of indirection in software to expose a virtual resource or device that corresponds to a physical device, even if it is not present in the current computer system. Cross-architectural emulators run one processor architecture on another, e.g., Apple Rosetta emulates a PowerPC processor on an x86 computer for backward compatibility. In this example, X=PowerPC and Y=x86 in Figure 1.1c. The virtual abstraction corresponds to a particular processor with a well-defined ISA, even though the physical processor is different. Memory and disks can emulate each other: a RAM disk emulates the function of a disk using DRAM as backing store. The paging process of virtual memory does the opposite: the operating system uses disk sectors to emulate virtual memory.

Multiplexing, aggregation, and emulation can naturally be combined together to form a complete execution stack. In particular, as we will see shortly in §1.5, nearly all hypervisors incorporate a combination of multiplexing and emulation.

1.2 VIRTUAL MACHINES

The term "virtual machine" has been used to describe different abstractions depending on epoch and context. Fortunately, all uses are consistent with the following, broad definition.

> A **virtual machine** is a complete compute environment with its own isolated processing capabilities, memory, and communication channels.

This definition applies to a range of distinct, incompatible abstractions, illustrated in Figure 1.2:

- **language-based virtual machines**, such as the Java Virtual Machine, Microsoft Common Language Runtime, Javascript engines embedded in browsers, and in general the run-time environment of any managed language. These runtime environments are focused on running single applications and are not within the scope of this book;

- **lightweight virtual machines**, which rely on a combination of hardware and software isolation mechanisms to ensure that applications running directly on the processor (e.g., as native x86 code) are securely isolated from other sandboxes and the underlying operating system. This includes server-centric systems such as Denali [182] as well as desktop-centric systems such as the Google Native Client [190] and Vx32 [71]. Solutions based on Linux containers such as Docker [129] or the equivalent FreeBSD Jail [110] fall into the same category. We will refer to some of these systems as applicable, in particular in the context of the use of particular processor features; and

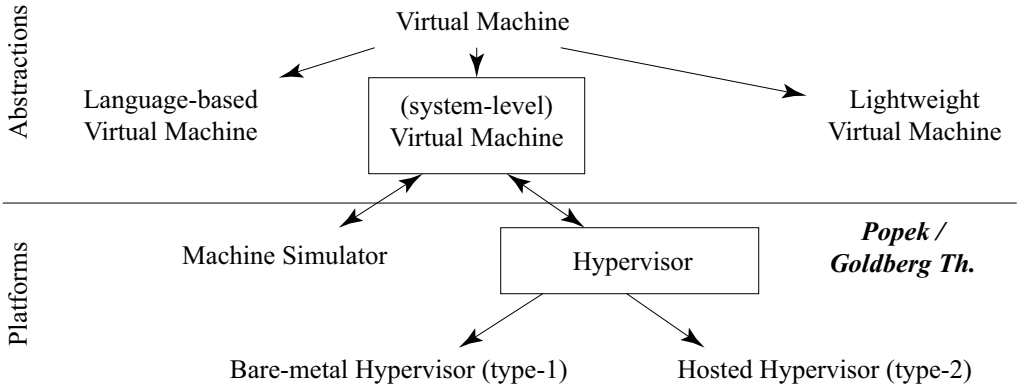

Figure 1.2: Basic classification of virtual machines and the platforms that run them.

- **system-level virtual machines**, in which the isolated compute environment resembles the hardware of a computer so that the virtual machine can run a standard, commodity operating system and its applications, in full isolation from the other virtual machines and the rest of the environment. Such virtual machines apply the virtualization principle to an *entire computer system*. Each virtual machine has its own copy of the underlying hardware, or at least, its own copy of *some* underlying hardware. Each virtual machine runs its own independent operating system instance, called the **guest operating system**. This is the essential focus of this book.

Figure 1.2 also categories the various platforms that run system-level virtual machines. We call these platforms either a hypervisor or a machine simulator, depending on the techniques used to run the virtual machine:

- a **hypervisor** relies on **direct execution** on the CPU for maximum efficiency, ideally to eliminate performance overheads altogether. In direct execution, the hypervisor sets up the hardware environment, but then lets the virtual machine instructions execute directly on the processor. As these instruction sequences must operate within the abstraction of the virtual machine, their execution causes traps, which must be emulated by the hypervisor. This **trap-and-emulate** paradigm is central to the design of hypervisors; and

- a **machine simulator** is typically implemented as a normal user-level application, with the goal of providing an accurate simulation of the virtualized architecture, and often runs at a small fraction of the native speed, ranging from a 5× slowdown to a 1000× slowdown, depending on the level of simulation detail. Machine simulators play a fundamental role in computer architecture by allowing the detailed architectural study of complex workloads [36, 124, 153, 181].

1.3 HYPERVISORS

A **hypervisor** is a special form of system software that runs virtual machines with the goal of minimizing execution overheads. When multiple virtual machines co-exist simultaneously on the same computer system, the hypervisor multiplexes (i.e., allocates and schedules) the physical resources appropriately among the virtual machines.

Popek and Goldberg formalized the relationship between a virtual machine and hypervisor (which they call VMM) in 1974 as follows [143].

> A virtual machine is taken to be an **efficient, isolated duplicate** of the real machine. We explain these notions through the idea of a virtual machine monitor (VMM). As a piece of software, a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs running in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.

Popek and Goldberg's definition is consistent with the broader definition of virtualization: the hypervisor applies the layering principle to the computer with three specific criteria of equivalence, safety and performance.

Equivalence: Duplication ensures that the exposed resource (i.e., the virtual machine) is equivalent with the underlying computer. This is a strong requirement, which has been historically relaxed in some measure when the architecture demands it (see §1.7).

Safety: Isolation requires that the virtual machines are isolated from each other as well as from the hypervisor, which enforces the modularity of the system. Critically, the safety of the design is enforced by the hypervisor without it making any assumptions about the software running inside the virtual machine (including the guest operating system).

Performance: Finally, and critically, Popek and Goldberg's definition adds an additional requirement: the virtual system must *show at worst a minor decrease in speed*. This final requirement separates hypervisors from machine simulators. Although machine simulators also meet the duplication and the isolation criteria, they fail the efficiency criteria as even fast machine simulators using dynamic binary translation [32, 124, 153, 184] slow down the execution of the target system by at least 5×, in large part because of the high cost of emulating the TLB of the virtual machine in software.

1.4 TYPE-1 AND TYPE-2 HYPERVISORS

Finally, Figure 1.2 shows that hypervisor architectures can be classified into so-called **type-1** and **type-2**. Robert Goldberg introduced these terms in his thesis [77], and the terms have been used ever since. Informally, a type-1 hypervisor is in direct controls of all resources of the physical computer. In contrast, a type-2 hypervisor operates either "as part of" or "on top of" an existing host operating system. Regrettably, the literature has applied the definitions loosely, leading to some confusion. Goldberg's definition (using an updated terminology) is as follows.

> The implementation requirement specifies that instructions execute directly on the host. It does not indicate how the hypervisor gains control for that subset of instructions that must be interpreted. This may be done either by a program running on the bare host machine or by a program running under some operating system on the host machine. In the case of running under an operating system, the host operating system primitives may be used to simplify writing the virtual machine monitor. Thus, two additional VMM categories arise:
>
> - **type-1:** the VMM runs on a bare machine;
> - **type-2:** the VMM runs on an extended host, under the host operating system.
>
> [...] In both type-1 and type-2 VMM, the VMM creates the virtual machine. However, in a type-1 environment, the VMM on a bare machine must perform the system's scheduling and (real) resource allocation. Thus, the type-1 VMM may include such code not specifically needed for virtualization. In a type-2 system, the resource allocation and environment creation functions for virtual machine are more clearly split. The operating system does the normal system resource allocation and provides a standard extended machine.

We note that the emphasis is on resource allocation, and not whether the hypervisor runs in privileged or non-privileged mode. In particular, a hypervisor can be a type-2 even when it runs in kernel-mode, e.g., Linux/KVM and VMware Workstation operate this way. In fact, Goldberg assumed that the hypervisor would always be executing with supervisor privileges.

Both types are commonly found in current systems. First, VMware ESX Server [177], Xen [27, 146], and Microsoft Hyper-V are all type-1 hypervisors. Even though Xen and Hyper-V depend on a host environment called *dom0*, the hypervisor itself makes the resource allocation and scheduling decisions.

Conversely, VMware Workstation [45], VMware Fusion, KVM [113], Microsoft VirtualPC, Parallels, and Oracle VirtualBox [179] are all type-2 hypervisors. They cooperate with a

host operating system so that the host operating system schedules all system resources. The host operating system schedules the hypervisor as if it were a process, even though these systems all depend on a heavy kernel-mode component to execute the virtual machine. Some hypervisors such as VMware Workstation and Oracle VirtualBox are portable across different host operating systems, while Fusion and Parallels runs with the Mac OS X host operating system, Microsoft Virtual PC runs with the Windows host operating system and KVM runs as part of the Linux host operating system. Among these type-2 systems, KVM provides the best integration with the host operating system, as the kernel mode component of the hypervisors is integrated directly within the Linux host as a kernel module.

1.5 A SKETCH HYPERVISOR: MULTIPLEXING AND EMULATION

With the basic definitions established, we now move to a first sketch description of the key elements of a virtualized computer system, i.e., the specification of a virtual machine and the basic building blocks of the hypervisor.

Figure 1.3 illustrates the key architectural components of a virtualized computer system. The figure shows three virtual machines, each with their own virtual hardware, their own guest operating system, and their own applications. The hypervisor controls the actual physical resources and runs directly on the hardware. In this simplified architecture, the hardware (virtual or physical) consists of processing elements, which comprises one or more CPUs, their MMU, and cache-coherent memory. The processing elements are connected to an I/O bus, with two attached I/O devices: a disk and a network interface card in Fig 1.3. This is representative of a server deployment. A desktop platform would include additional devices such as a keyboard, video, mouse, serial ports, USB ports, etc. A mobile platform might further require a GPS, an accelerometer, and radios.

In its most basic form, a hypervisor uses two of the three key virtualization techniques of §1.1: it *multiplexes* (in space, and possibly in time) the physical PE across the virtual machines, and it *emulates* everything else, in particular the I/O bus and the I/O devices.

This combination of techniques is both necessary and sufficient in practice to achieve the efficiency criteria. It is necessary because without an effective mechanism to multiplex the CPU and the MMU, the hypervisor would have to emulate the execution of the virtual machine. In fact, the principal difference between a machine simulator and a hypervisor is that the former emulates the virtual machine's instruction set architecture, while the later multiplexes it. Multiplexing of the CPU is a scheduling task, very similar to the one performed by the operating system to schedule processes. The scheduling entity (here, the hypervisor) sets up the hardware environments (register file, etc.) and then lets the scheduled entity (the virtual machine) run directly on the hardware with reduced privileges.

This scheduling technique is known as **direct execution** since the hypervisor lets the virtual CPU directly execute instructions on the real processor. Of course, the hypervisor is also

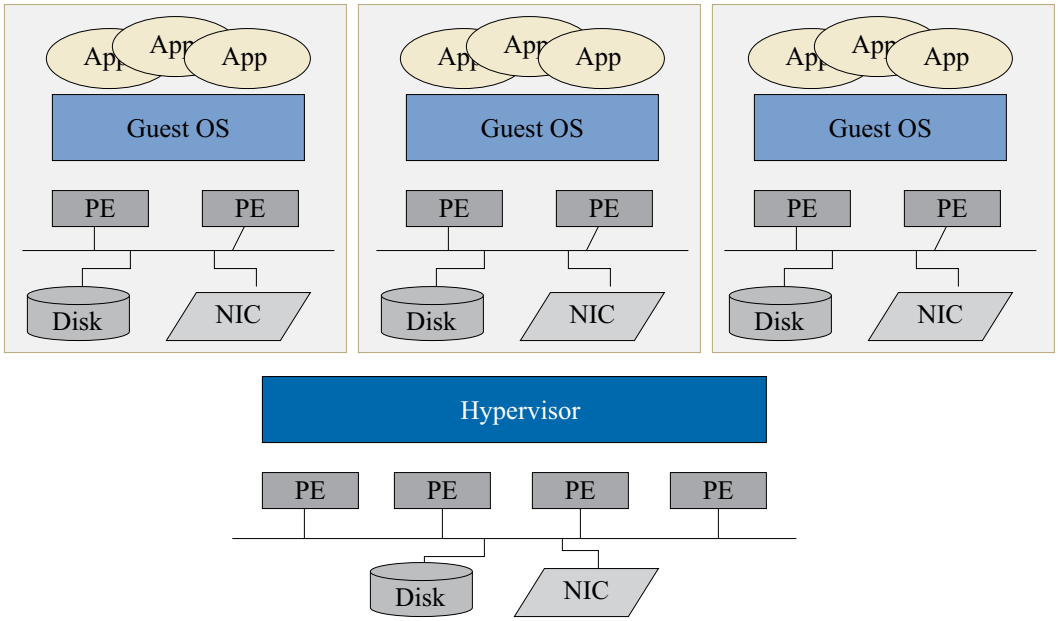

Figure 1.3: Basic architecture of a virtual machine monitor. Each processing element (PE) consists of CPU and physical memory.

responsible to ensure the safety property of the virtual machine. It therefore ensures that the virtual CPU always executes with reduced privileges, e.g., so that it cannot execute privileged instructions. As a consequence, the direct execution of the virtual machine leads to frequent traps whenever the guest operating system attempts to execute a privileged instruction, which must be emulated by the hypervisor. Hypervisors designed around direct execution therefore follow a **trap-and-emulate** programming paradigm, where the bulk of the execution overhead is due to the hypervisor emulating traps on behalf of the virtual machine.

Physical memory is also multiplexed among the virtual machines, so that each has the illusion of a contiguous, fixed-size, amount of physical memory. This is similar to the allocation among processes done by an operating system. The unique challenges in building a hypervisor lie in the virtualization of the MMU, and in the ability to expose user-level and kernel-level execution environment to the virtual machine.

The combination of multiplexing and emulation is also sufficient, as I/O operations of today's computer systems are implemented via reasonably high-level operations, e.g., a device driver can issue simple commands to send a list of network packets specified in a descriptor ring, or issue a 32 KB disk request. A hypervisor emulates the hardware/software interface of at least one representative device per category, i.e., one disk device, one network device, one screen device,

etc. As part of this emulation, the hypervisor uses the available physical devices to issue the actual I/O.

I/O emulation has long been the preferred approach to the virtualization of I/O device because of its portability advantages: a virtual machine "sees" the same virtual hardware, even when running on platform with different hardware devices. Today, modern hardware includes advanced architectural support for I/O virtualization which enables the multiplexing of certain classes of I/O devices, with notable performance benefits in terms of throughput and latency, but still at the expense of reducing the mobility and portability of the virtual machines.

Table 1.1 provides a concrete example of the combined use of multiplexing and emulation in VMware Workstation 2.0, an early, desktop-oriented hypervisor, released in 2000. Clearly, the hardware is dated: USB is notably missing, and most readers have never seen actual floppy disks. The concepts however remain the same. For each component, Table 1.1 describes the **front-end device** abstraction, visible as hardware to the virtual machine, and the **back-end emulation** mechanisms used to implement it. When a resource is multiplexed, such as the x86 CPU or the memory, the front-end and back-end are identical, and defined by the hardware. The hypervisor is involved only to establish the mapping between the virtual and the physical resource, which the hardware can then directly use without further interception.

Table 1.1: Virtual Hardware of early VMware Workstation [45]

	Virtual Hardware (front-end)	Back-end
Multiplexed	1 virtual x86-32 CPU	Scheduled by the host operating system with one or more x86 CPUs
	Up to 512 MB of contiguous DRAM	Allocated and managed by the host OS (page-by-page)
Emulated	PCI Bus	Fully emulated compliant PCI bus with B/D/F addressing for all virtual mother-board and slot devices
	4 x 4IDE disks 7 x Buslogic SCSI Disks	Either virtual disks (stored as files) or direct access to a given raw device
	1 x IDE CD-ROM	ISO image or real CD-ROM
	2 x 1.44 MB floppy drives	Physical floppy or floppy image
	1 x VGA/SVGA graphics card	Appears as a Window or in full-screen mode
	2 x serial ports COM1 and COM2	Connect to Host serial port or a file
	1 x printer (LPT)	Can connect to host LPT port
	1 x keyboard (104-key) and mouse	Fully emulated
	AMD PCnet NIC (AM79C970A)	Via virtual switch of the host

When the resource is emulated, however, the **front-end device** corresponds to one canonically chosen representative of the device, independent of the back-end. The hypervisor implements both the front-end and the back-end, typically in software without any particular hardware support. The front-end is effectively a software model of the chosen, representative device. The **back-end emulation** chooses among the underlying resources to implement the functionality. These underlying resources may be physical devices or some higher-level abstractions of the host operating system. For example, the disk front-ends in VMware Workstation were either IDE or Buslogic SCSI devices, two popular choices at the time, with ubiquitous device drivers. The backend resource could be either a physical device, i.e., an actual raw disk, or a virtual disk stored as a large file within an existing filesystem.

Although no longer an exact duplicate of the underlying hardware, the virtual machine remains compatible. Assuming that a different set of device drivers can be loaded inside the guest operating system, the virtual machine will have the same functionality.

So far, this hypervisor sketch assumes that the various components of the processing elements can be virtualized. Yet, we've also alluded to the historical fact that some hardware architectures fail to provide hardware support for virtualization. This discussion will be the core of Chapter 2 and Chapter 3.

1.6 NAMES FOR MEMORY

The cliché claims (apparently incorrectly according to linguists) that Eskimos have many names for snow. Similarly, computer architecture and system designers have used at times overlapping, somewhat confusing definitions for the many facets of memory. The reason is simple. Like snow to Eskimos, virtual memory is fundamental to operating systems and arguably the most significant enhancements over the original von Neuman model of computing in this context. In reality, there are fundamentally only two types of memory: **virtual memory** converts into **physical memory** via the combination of segmentation and paging.

Virtual memory: Virtual memory refers to the byte-addressable namespace used by instruction sequences executed by the processor. With rare exceptions, all registers and the instruction pointer that refer to a memory location contain a **virtual address**. In a segmented architecture, the virtual address space is determined by a base address and a limit. The former is added to the virtual address and the latter is checked for protection purposes. In a paging architecture, the virtual address space is determined by the memory management unit on a page-by-page basis, with the mapping defined either by page tables or by a software TLB miss handler. Some architectures such as x86-32 combine segmentation with paging. The virtual address is first converted (via segmentation) into a **linear address**, and then (via paging) into a physical address.

Physical memory: Physical memory refers to the byte-addressable resource accessed via the memory hierarchy of the processor, and typically backed by DRAM. In a non-virtualized computer system, the **physical address space** is generally determined by the resources of the hardware,

and defined by the memory controller of the processor. In a virtualized computer system, the hypervisor defines the amount of physical memory that is available to the virtual machine. Technically, the abstraction visible to the virtual machine should be called "virtual physical memory." However, to avoid confusion with virtual memory, it is most commonly called **guest-physical memory**. To further avoid ambiguity, the underlying resource is called **host-physical memory**. Note that some early virtualization papers used different terminologies, in particular using the terms *physical memory* and **machine memory** to refer to guest-physical and host-physical memory, respectively.

1.7 APPROACHES TO VIRTUALIZATION AND PARAVIRTUALIZATION

The lack of clear architectural support for virtualization in earlier processors has led to a range of approaches that solve the identical problem of running a virtual machine that is either similar or compatible with the underlying hardware. This text covers the three pragmatic approaches to virtualization.

Full (software) virtualization: This refers to hypervisors designed to maximize hardware compatibility, and in particular run unmodified operating systems, on architectures lacking the full support for it. It includes in particular early versions of VMware's hypervisors. This is also referenced as software virtualization in some papers. We will describe the approach in §3.2.

Hardware Virtualization (HVM): This refers to hypervisors built for architectures that provide architectural support for virtualization, which includes all recent processors. Such hypervisors also support unmodified guest operating systems. Unlike software virtualization approach in which the hypervisor must (at least some of the time) translate guest instruction sequences before execution, HVM hypervisors rely exclusively on **direct execution** to execute virtual machine instructions. In the literature, HVM is at times referred to as HV. The requirements for an HVM are formally defined in Chapter 2. Architectures and hypervisors that follow the approach are the focus of Chapter 4 for x86 with VT-x and Chapter 7 for ARM with Virtualization Extensions.

Paravirtualization: This approach makes a different trade-off, and values simplicity and overall efficiency over the full compatibility with the underlying hardware. The term was introduced by Denali [182] and popularized by the early Xen hypervisor. In its original usage on platforms with no architectural support for virtualization, paravirtualization required a change of the guest operating system binary, which was incompatible with the underlying hardware. In its contemporary use on architectures with full virtualization support, paravirtualization is still used to augment the HVM through platform-specific extensions often implemented in device drivers, e.g., to manage cooperatively memory or to implement a high-performance front-end device. We will describe this in §3.3.

1.8 BENEFITS OF USING VIRTUAL MACHINES

So far, we have discussed the "how" of virtualization as any good engineering textbook should, but not the "why?", which is equally relevant. Virtual machines were first invented on mainframes when hardware was scarce and very expensive, and operating systems were primitive. Today, they are used very broadly because of fundamentally different considerations. We enumerate some of the most popular ones.

Operating system diversity: Virtualization enables a diversity of operating systems to run concurrently on a single machine. This benefit is the key reason behind many desktop-oriented type-2 hypervisors, e.g., Fusion and Parallels are essentially used to run Windows (and sometimes Linux) on Mac OS X.

Server consolidation: Enterprise IT best practices today still mandate that each server runs a single application per machine. With the rise in power and efficiency of hardware, that machine is today more often than not a virtual machine [95].

Rapid provisioning: Deploying a physical server is complex, time-intensive task. In contrast, a virtual machine can be created entirely in software through a portal or an API, and software stack can be deployed as virtual appliances [157].

Security: Virtualization introduces a new level of management in the datacenter stack, distinct and invisible from the guest operating systems, and yet capable of introspecting the behavior of such operating systems [49], performing intrusion analysis [68], or attesting of its provenance [73]. The hypervisor can also control all I/O operations from the virtual machine, making it easy to insert e.g., a VM-specific firewall or connect into a virtual network [114].

High-availability: A virtual machine is an encapsulated abstraction that can run on any server (running a compatible hypervisor). Specifically, a virtual machine can reboot following a hardware crash on a new server without operational impact, therefore providing a high-availability solution without requiring guest operating system-level configuration or awareness.

Distributed resource scheduling: The use of live migration technologies turns a cluster of hypervisors into a single resource pools, allowing the automatic and transparent rebalancing of virtual machines within the cluster [96].

Cloud computing: In a virtualized environment, different customers (tenants) can operate their own virtual machines in isolation from each other. When coupled with network virtualization (a technology outside of the scope of this text), this provides the foundation for the cloud computing technologies, including the ones of Amazon Web Services, Google Compute Engine, and Microsoft Azure.

1.9 FURTHER READING

We rely largely on Salzer and Kaashoek's book, *Principles of Computer Systems* [155], for definitions of layering and enforced modularity. The book provides excellent further reading to readers interested in additional background material, or who wish to look at other examples of virtualization. Tanenbaum and Bos' *Modern Operating Systems*, 4th ed. [165] also dedicates a chapter to virtualization.

CHAPTER 2

The Popek/Goldberg Theorem

In 1974, Gerald Popek and Robert Goldberg published in *Communications of the ACM* the seminal paper "Formal Requirements for Virtualizable Third-Generation Architectures" that defines the necessary and sufficient formal requirements to ensure that a VMM can be constructed [143]. Precisely, their theorem determines whether a given **instruction set architecture (ISA)** can be virtualized by a VMM using multiplexing. For any architecture that meets the hypothesis of the theorem, any operating system directly running on the hardware can also run inside a virtual machine, without modifications.

At the time, the motivation for the work was to address the evidence that new architectures accidentally prevented the construction of a VMM. The authors cited the DEC PDP-10 in their article, where seemingly arbitrary architectural decisions "broke" virtualization. Despite the simplicity of the result, the relevance of the theorem was lost on computer architects for decades, and generations of new processor architectures were designed without any technical considerations for the theorem.

Much later, as virtual machines once again became relevant, Intel and AMD explicitly made sure that their virtualization extensions met the Popek and Goldberg criteria, so that unmodified guest operating systems could run directly in virtual machines, without having to resort to software translation or paravirtualization [171].

Today, the theorem remains the obvious starting point to understand the fundamental relationship between a computer architecture and its ability to support virtual machines. Specifically, the theorem determines whether a VMM, relying exclusively on direct execution, can support guest arbitrary operating systems.

2.1 THE MODEL

The paper assumes a standard computer architecture, which the authors call a **conventional third-generation architecture**. The processor has two execution modes (user-level and supervisor), and support for virtual memory. Such an architecture is both necessary and sufficient to run a conventional operating system. In particular, the operating system can configure the hardware to run multiple, arbitrary, potentially malicious applications in isolation from each other. For the purpose of the proof, the paper defines a simple model that remains representative of the broader class of these architectures. Specifically, the model has the following features.

- The computer has one processor with two execution levels: supervisor mode and user mode.